Modelling Periodic Temporal Notions by Labelled Partitionings of the Real Numbers – The PartLib Library

Hans Jürgen Ohlbach
Institut für Informatik, Universität München
Oettingenstr. 67
D-80538 München, Germany
email: ohlbach@lmu.de

Abstract

The key notion for modelling calendar systems and many other periodic temporal notion is the mathematical concept of *a partitioning of the real numbers*. A partitioning of $\mathbb{R}$ splits the time axis into a sequence of intervals. Basic time units like seconds, minutes, hours, days, weeks, months, years etc. can all be represented by partitionings of $\mathbb{R}$ with finite partitions. Besides the basic time units in calendar systems, there are a lot of other temporal notions which can be modelled as partitions: the seasons, the ecclesiastical calendars, financial years, semesters at universities, the sequence of sunrises and sunsets, the sequence of the tides, the sequence of school holidays etc. In this paper a formalization of periodic temporal notions by means of *labelled partitionings* of $\mathbb{R}$ is presented. The formalism is implemented as the C++ library *PartLib* (Partitioning Library). The interface to PartLib is presented in the appendix.

keywords: calendrical calculations, representation of time, ontology of temporal notions, time granularities, knowledge representation,.

Contents

1 Motivation and Introduction **2**

2 Partitionings **7**
2.1 Length of Intervals in Partitions 9
2.2 Labels 10
2.3 Granules 10
2.4 Relations Between Partitionings 12

3 Date Formats **13**

4 Shift Functions **14**
4.1 Length Oriented Shift Function 14
4.2 Date Oriented Shift Function 15
4.3 Shift by Granules 16

5 Durations **18**

6 Calendar Systems **19**

7 Global and Local Reference Time **19**

8 Specification of Partitionings **20**
8.1 Algorithmic Partitionings 21
8.2 Duration Partitionings 23
8.3 Regular Partitionings 24
8.4 Date Partitionings 26
8.5 Folded Partitionings 27

9 Service Functions **30**
9.1 Which 30

10 Summary **31**

A The PartLib Interface **31**
A.1 Repetition Patterns 32
A.2 The Label Class 32
A.3 The Labelling Class 33
A.4 The Duration Class 35
A.5 Class DateFormat 35
A.6 Class DateData 36
A.7 The Partitioning Class 36
A.7.1 The Abstract Class 'Partitioning' 37
A.7.2 Subclasses of the Partitioning class 40

1 Motivation and Introduction

The basic time units of calendar systems, years, months, weeks, days etc. are the prototypes of periodic temporal notions. Because time is one of the most important parameters of our life, the representation of temporal notions, and in particular periodic temporal notions, is necessary in many computer applications. There have been quite intensive studies of periodic temporal notions from various points of view. One can distinguish at least three approaches.

First of all, there is the important work of Dershowitz and Reingold [9] who analyzed existing calendar systems and came up with algorithms for converting date information from one system to another. These algorithms are the basis for the implementation of concrete calendar systems in computer programs.

On a more abstract level there is all the work about the mathematical representation of periodic temporal notions as *time granularities*, or similar kind of mathematical objects. A good overview is given in the book of Bettini, Jajoda and Wang [3]. This work is particularly motivated by the need to represent time in temporal databases. A selection of papers about the abundant work in this area is [1, 17, 13, 21, 14, 16, 10, 2, 11, 4, 12, 5]. Since time granularities are the most important objects in this area, we introduce them already at this early place in the paper. A time granularity is usually defined as a mapping of a subset of the integers to sets of intervals in the time domain, the *granules*. This mapping must have certain properties in order to count as time granularity. Another way to explain time granularities is: a *granule* is a, possibly non-convex finite subinterval of the time domain. A *time granularity* is a sequence of such granules. One can require that this sequence is consecutive, i.e. the rightmost time point of a granule n comes before the leftmost time point of the granule $n + 1$. Sometimes, however, overlapping granules are also considered [11]. The simplest time granularities are in fact partitionings of the time domain. All basic time units, years, months etc., are of this type. The granules consist of one single interval, and there are no gaps between them. Granules consisting of one single interval only, but with gaps between them, can, for example, be used to model 'weekend'. The time spans between the weekends are the gaps between the granules. Granules consisting of several intervals are useful to model notions like 'my working day', where there is

a lunch break which should not count as part of 'my working day'. Overlapping granules might be used to model, for example, the union of 'my working day' and 'my wife's working day'. The 'time granularity community' has developed ways for constructing time granularities, usually as algebraic operations on previously constructed time granularities. Conversion operations between different granularities have been defined. Relations between different time granularities have been developed, and applications, mainly in the area of temporal databases, have been considered.

An even further abstraction is possible by axiomatizing temporal notions in an expressive enough logic, for example in first order predicate logic. The SOL time theory (SOL for Structured Temporal Object) of Diana Cuckierman with a first order formalization of time loops is a prominent example for this approach [6, 7, 8].

This paper presents an alternative to the granularities approach. We represent periodic temporal notions as partitionings of the real numbers, which is the simplest form of granularities. To compensate for this very weak structure, we introduce names (labels) for the partitions. The labels carry information about the meaning of the partitions. As we shall see, this separation of structure and meaning has a number of algorithmic advantageous. A built-in label is 'gap'. It can be used to denote a partition which logically does not belong to a given partitioning. For example, the time between two subsequent school holidays in a school holiday partitioning can be labelled 'gap'. The label 'gap' allows one to simulate the granules of time granularities and nevertheless keep the algorithmic advantages.

The CTTN System

The work on partitionings for modelling periodic temporal notions is part of the CTTN-project. The CTTN system (Computational Treatment of Temporal Notions) [18] is a system for understanding, representing and manipulating complex temporal notions from everyday life. The basic modules in the CTTN system are the FuTI library for representing and manipulating fuzzy time intervals [19], the PartLib library for representing and manipulating periodic temporal notions, different calendar systems, and finally the GeTS language [20]. The GeTS language (GeoTemporal Specifications) is a typed functional programming language with a lot of built-in data types and operations for manipulating temporal notions. The GeTS language has in particular access to the FuTI library for dealing with crisp and fuzzy time intervals, and to the PartLib library for dealing with periodic temporal notions. A simple example for a definition in GeTS is

$$tomorrow = partition(now(), day, 1, 1).$$

$now()$ yields the current moment in time, measured in seconds. *day* refers to the day partitioning from the currently activated calendar, $partition(\ldots, day, 1, 1)$ creates the interval that corresponds to the day partition of the next day. The algorithms for computing the boundaries of this interval are presented in this document.

Remark 1.1 *It is important to notice that the ideas and techniques presented in this paper are only one piece in a bigger mosaic. The labelled partitionings together with the operations described in this paper are only one of several components in the GeTS language, which is the main tool for representing and working with temporal notions. Not all operations which are in principle possible with partitionings are therefore realized in the PartLib library itself, but on another level of the CTTN–system. In particular, there is no direct support for logical inferencing with the partitionings in the PartLib library.* ∎

Guidelines

The guidelines for the particular approach presented in this paper, and realized in the PartLib library were:

1. The reality should be taken serious:
This means that all phenomena in real calendar systems and realistic periodic temporal notions should be taken into account. The consequences of this can be illustrated with the following definition of *month* taken from [17]: The authors defined `day` first, then
$\texttt{pseudomonth} = Alter^{12}_{11,-1}(day, Alter^{12}_{9,-1}(day, Alter^{12}_{6,-1}(day, Alter^{12}_{4,-1}($
$day, Alter^{12}_{2,-3}(day, Group_{31}(day))))))$
and finally
$\texttt{month} = Alter^{12\cdot 400}_{2+12\cdot 399,1}(day, Alter^{12\cdot 100}_{2+12\cdot 99,-1}(day, Alter^{12\cdot 4}_{2+12\cdot 3,1}(day, \texttt{pseudomonth})))$.

The last definition takes leap days into account. *Alter* is the *alternating tick* operator and *Group* is the grouping operator. It is not necessary here to understand these operators. The point is that in a user friendly implementation of these operators an evaluator for arithmetic expressions like $2 + 12 \cdot 3$ is needed. This should be be no problem as long as the expressions are simple enough. For more complex temporal notions, however, the expressions become also more complex, and eventually a full size programming language is needed here. For example, for modelling ecclesiastical calendars, one needs to calculate the Easter date, and the algorithm for this is too complex to be expressed as a simple arithmetic formula.

The consequence for the PartLib library was to introduce a partitioning type *algorithmic partitionings*, which is specified by providing concrete algorithms in a concrete programming language (C++ in this case). Nevertheless, this is an exception. The general guideline is 'as algorithmic as necessary, as symbolic as possible'. The algorithmic partitionings can be used to define what some authors call *basic calendars* [15, 11].

2. Separation of structure and meaning:
An infinite sequence of non-overlapping granules is in principle also a partitioning of the time domain if the gaps between the granules and within the granules are considered as part of the partitioning. Therefore one can turn a partitioning into a granularity by labelling certain partitions as gaps, and labelling the partitions which should belong to a granule with a common name. This has the advantages that the algorithms can be separated into a part which deals with the structure of the partitions, and a part which deals with the labels. Moreover, the labels can be used for other purposes. For example, the labels of a bus timetable can be the bus identifiers, and these can be keys for a bus database.

Notice that the notion of a 'label' in this paper is different to the notion of a 'label' in the literature about granularities. Labels in this paper are *names*, i.e. strings like 'Monday', 'Tuesday' etc. The 'labels' in the literature about granularities correspond to *coordinates* in this paper.

PartLib provides no means for representing overlapping granularities as a single object. They must be represented as two separate partitionings.

3. Compact data structures and efficient algorithms:
The partitionings must be represented with finite data structures which support a number of particular algorithms. This excludes certain problematic operators for constructing new partitionings (granularities) from existing ones. An example is the *union* operator. To understand this, consider a representation of, say, 'Tom's working day' and 'Jane's working day'. If Tom's working day is every day from 8 am until 5 pm, and 'Jane's working day' is every day from 9 am until 6 pm, then the union operation on these two partitionings is unproblematic. The resulting partitioning is easily representable in a compact way. If, however, Tom's job is to watch the moon in an observatory, then his working day may need to follow the moon phases, which can be represented with an algorithmic partitioning. The union of the two partitionings 'Tom's working day' and 'Jane's working day' is now a really complicated object, and not easily represented. Therefore we exclude operations like union, intersection etc. in PartLib itself. Instead we provide such operations in the GeTS language. What is easy to realize is an operation which cuts 'Tom's working days' and 'Jane's working day' out of a given *finite* interval and then applies the set operation to the two intervals. Therefore 'Tom's and Jane's working day' would

not be represented as a partitioning, but as a function that takes a time interval I and returns the subintervals of I which corresponds to the union of Tom's working days and Jane's working days.

4. Intuitive specification of user defined partitionings:
Most basic time units of calendar systems have a non-trivial algorithmic component. In the CTTN–system they are therefore realized as built-in partitionings. Many others, however, are application specific or user defined. Therefore various authors have come up with algebraic operations for constructing new partitionings (granularities) from given ones [11]. The art is to find a basic set of operations which allows one to define new partitionings in a way which is intuitive to the user, and which provides good data structures for the algorithms. This set should be as small as possible in order to reduce the burden to develop the corresponding algorithms. On the other hand, it should be powerful and expressive enough that most real world examples for periodic temporal notions can be specified.

Besides the algorithmic partitionings, PartLib provides two more basic types of specifications: 'duration partitionings' and 'folded partitionings'. Duration partitionings are specified by an anchor time and a sequence of 'durations'. A *duration* is something like '1 month + 3 day', where 'month' and 'day' represent previously defined partitionings. For example, I could define 'my weekend' as a *duration partitioning* with anchor time 2004/7/23, 4 pm (Friday July, 23rd, 2004, 4 pm) and durations: ('8 hour + 2 day', '4 day + 16 hour'). The first interval would be labelled 'weekend', and the second interval would be labelled 'gap'.

Notice that this specification is different to ('56 hour', '112 hour'). The difference is that when standard time changes to daylight savings time then the day is only 23 hours long, and when daylight savings time changes to standard time then the day is 25 hours long. A proper representation of 'day' as an algorithmic partitioning can take this into account, such that '8 hour + 2 day' would be the correct time shift even in this case. The specification of 'weekend' with a duration of '56 hour', however, would become wrong during the daylight savings time period.

PartLib provides two specializations of duration partitionings. They allow for faster algorithms, and they are more intuitive in certain cases. The first specialization, the *regular partitionings* covers the case that the durations are all of the same kind 'n P' where P is always the same partitioning. For example, 'semester' could be defined this way. The anchor time is the start of, say, the winter semester. The durations are ('6 month', '6 month'). The first partition would be labelled 'winter semester', and the second partition would be labelled 'summer semester'. The algorithms for this simple kind of duration partitioning are more efficient than in the general case.

Another specialization are *date partitionings*. In this version the partitions are specified by concrete dates. In many countries, for example, people used to count the years from the beginning of the reigns of their emperors, and these are concrete dates. At a first glance, this specifies a partitioning of only a finite part of the time domain. To take this into account many of the algorithms would need to check whether the time points under consideration are in the valid part of the time line where the partitioning is specified, or not. With a simple trick, however, one can turn this finite partitioning into an infinite partitioning, and thus avoid these special cases. The trick is to turn the difference between two consecutive dates into durations. For example, the two dates 2004/5/10 and 2006/8/15 can be turned into a duration '2 year + 3 month + 5 day'. This way a date partitioning is turned into a duration partitioning. The finite part of the date partitioning is then automatically extrapolated into the infinite future and past. PartLib provides means to define boundaries for the partitionings, but these boundaries are not checked by the algorithms. It is up to the application of PartLib to check the boundaries.

Duration partitionings are the second basic type of partitionings. The third type are *folded partitionings*. Consider a bus timetable, which changes from season to season. The best way to specify this, would be to specify the seasons first, and for each season to specify the particular bus

timetable. The 'folded partitioning' specification operation takes as input a *frame partitioning*, for example the seasons, and a sequence of *folded partitionings*, for example the four different bus timetables. It maps the folded partitionings automatically to the right frame partition, such that from the outside the whole thing looks like an ordinary partitioning.

5. Support for certain key operations with partitionings:
A very natural operation is to measure the distance between two time points in terms of a given time unit. 'The distance between t_1 and t_2 is 3.5 weeks', could, for example, be a useful information. Measuring the distance between two time points in terms of partitions of fixed length, for example seconds, is no point. It becomes more difficult if the time units have varying lengths. 'The distance between t_1 and t_2 is 3.5 months' is a nontrivial statement, because it depends on the location of t_1 and t_2 on the time line.

PartLib provides two *length* functions. The first one measure the distance between two time points in partitions of a given partitioning, and the second one measures the distance in granules. 'The distance between t_1 and t_2 is 1 working day', for example, is a possible outcome, even if 'working day' is defined not as a partitioning, but as a granule with a gap in it (for lunch time).

The second very natural operation is a shift operation, also in terms of partitions or granules. For example, one can ask a PartLib method to shift a time point t by, say 3.5 months, or 3.5 working days into the future. Since the lengths of the partitions and granules may vary, the concrete amount, t is shifted, depends on the location of t on the time line. It turns out that the notion of a time shift by some partitions is ambiguous. There are at least two different ways to do this, with more or less intuitive results. This problem is discussed in detail in Section 4.

PartLib offers some further operations. They are presented in the corresponding sections of this paper.

After a brief review of PartLib's time domain we present the formal definitions of the basic concepts and then discuss the specification mechanisms and the operations on partitionings. The interface to the PartLib implementation is presented in the appendix.

Time Measurements and the Semantics of Computer Time

The backbone of our time representation is a reference time line, measured in seconds. The relation between the artificial counting of seconds in the computer and the real flow of time on our planet is determined by the physics of time measurement. Before the adoption of the UTC standard (Coordinated Universal Time) in 1972, a second was just the 86400th fraction of a day, measured between two subsequent zeniths of the sun. Since the rotation of the earth is not perfectly stable over the year, and, moreover, slows down from year to year, these seconds corresponded to varying time intervals. After the adoption of the UTC standard, a second corresponds to exactly 9.192.631.770 cycles of the light emitted when an electron jumps between the two lowest hyperfine levels of the Cesium 133 atom (measured and coordinated by the 'Bureau International des Poids et Mesures' in Paris, URL: http://www.bipm.fr/). The synchronization with the rotation of the earth is achieved by inserting a leap second almost every year by the International Earth Rotation Service (URL: http://hpiers.obspm.fr/). Therefore the seconds in our modelling of partitionings for the time before 1972 correspond to a fraction of the day. For the time after 1972 they correspond to the atomic seconds of the TAI standard (Temps Atomique International).

In this paper it is assumed that there is a *global reference time GRT*, measured in seconds or some fraction of a second. *GRT* is actually isomorphic to the real numbers, but the number 0 corresponds to a particular point in time. As it is common in Unix systems, the origin of the reference time in our examples is the beginning of the year 1970 at the 0-meridian.

Remark 1.2 *Since the real numbers* $\mathbb{R}$ *are used as the time axis, we speak of the* earliest *or* leftmost *number, time point or interval if we mean the one closest to* $-\infty$*. We speak of the* latest *or* rightmost *number, time point or interval if we mean the one closest to* $+\infty$*.* ∎

2 Partitionings

A partitioning of the real numbers $\mathbb{R}$ may be for example $(..., [-100, 0[, [0, 100[, [100, 101[, [101, 500[, ...)$. The intervals in the partitionings considered in this paper need not be of the same length (because time units like years are not of the same length either). The intervals can, however, be enumerated by natural numbers (their *coordinates*). For example, we could have the following enumeration

...	$[-100$	$0[$	$[0$	$100[$	$[100$	$101[$	$[101$	$500[$	...
...		-1		0		1		2	...

It is not by chance that half open intervals are used in this example. Since the partitions in a partitioning do not overlap, one cannot use closed intervals because the endpoints of the closed intervals would be in two different partitions. Open intervals can not be used either because then the infima and suprema of the intervals would not be in any partition at all. Therefore only half open intervals can be used, either of the type $[a, b[$, or of the type $]a, b]$. In most cases there is no preference for either of the two types, but both types should not be used together. In this paper we therefore use the first type $[a, b[$.

Since all time measurements are done in discrete units (seconds, ticks of a clock, hyperfine transitions in a Cesium atom etc.) it makes sense to take integers as boundaries of the partitions. In the examples they represent seconds. Any other fraction of a second is possible as well. Multiples of seconds are not possible without loosing precision because the leap seconds are ignored in this case.

The formal definition for partitionings of $\mathbb{R}$ which is used in this paper is:

Definition 2.1 (Partitioning) *A partitioning P of the real numbers $\mathbb{R}$ is a sequence*

$$\ldots [t_{-1}, t_0[, [t_0, t_1[, [t_1, t_2[, \ldots$$

of non-empty half open intervals in $\mathbb{R}$ with integer boundaries. ∎

Some useful notations for partitionings are defined:

Definition 2.2 (Notations) *Let P be a partitioning.*

1. *For a partition $p = [s, t[$ in P let $p_[\stackrel{\text{def}}{=} s$ be left boundary of p and let $p_] \stackrel{\text{def}}{=} t$ be the right boundary of p.*
2. *For a time point t and a partitioning P, let t^P be the partition in P which contains t.* ∎

A sequence of finite partitions of the real numbers is in fact isomorphic to the integers. This can be exploited to give the partitions *addresses* or *coordinates*. The coordinates are very useful for navigating through sequences of partitions. Therefore we introduce *coordinate mappings*. In principle, there are many different coordinate mappings for a given partitioning, but for the intended application of the partitioning concept described in this paper, one single coordinate mapping is sufficient. Therefore this unique coordinate mapping becomes an integral component of a partitioning.

Definition 2.3 (Coordinate Mapping) *A* coordinate mapping *of a partitioning P is a bijective mapping between the intervals in P and the integers. Since we usually use one single coordinate mapping for a partitioning P, we can just use P itself to indicate the mapping.*
Therefore let p^P be the coordinate *of the partition p in P.*
For a coordinate i let i^P be the partition which corresponds to i.
For a time point t let $P.pc(t) \stackrel{\text{def}}{=} (t^P)^P$ be the coordinate of the partition containing t. (pc stands for 'partition coordinate').

Let $P.sopT(t) \stackrel{\text{def}}{=} t^P_{[}$ *be the start of the partition containing* t.
Let $P.eopT(t) \stackrel{\text{def}}{=} t^P_{[}$ *be the end of the partition containing* t.

For a coordinate i *let* $P.sopC(i) \stackrel{\text{def}}{=} i^P_{[}$ *be the start of the partition with coordinate* i.
Let $P.eopC(i) \stackrel{\text{def}}{=} i^P_{[}$ *be the end of the partition with coordinate* i.

For a time point t *we define*

$$P.lopT(t) \stackrel{\text{def}}{=} P.eopT(t) - P.sopT(t)$$

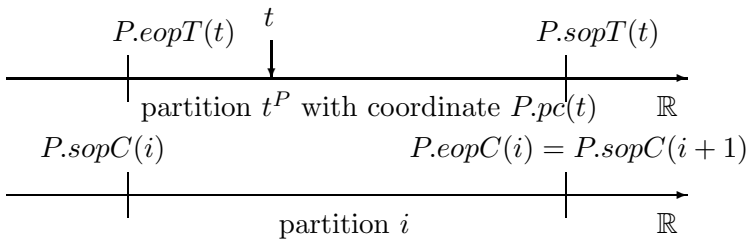
as the length of the partition containing t.

For a coordinate i *we define*

$$P.lopC(i) \stackrel{\text{def}}{=} P.lopT(P.sopC(i))$$

as the length of the partition with coordinate i. ■

The two pictures below illustrate the transitions between time points, coordinates and partitions:



Example 2.4 (Seconds and Minutes) *The partitioning for seconds consists of the sequence of intervals* $\ldots[-i,-i+1[\ldots[0,1[\ldots[k,k+1[\ldots$. *The interval* $[0,1[$ *represents the first second in January* 1^{st} *1970, and its coordinate is 0.*

The partitioning for minutes is

			1970/1/1 ↓			
$ref.time:$	...	$[-60,0[$	$[0,60[$	$[60,120[$	$[120,180[$	...
$coordinate:$	...	-1	0	1	2	,...

■

Remarks:

1. Partitions are not explictly represented in PartLib, only time points and coordinates. The functions which map time points to coordinates and back are therefore the important ones. Thus, the formulations of the algorithms below do not refer to partitions, but use these functions.

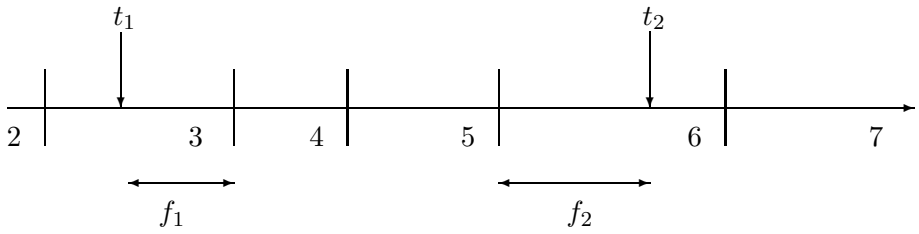
2. We use the notation $P.pc(t)$ for the function that maps a time point t to the coordinate of its partition in the partitioning P. Alternative notations would be $pc(P,t)$ or $pc_P(t)$ or $pc^P(t)$. The notation $P.pc(t)$ comes from object oriented programming. P is an object (instance of a class), and pc is a method in this class. This notation has two advantages. First of all, it is a bridge to the actual implementation where the program code looks just so. Secondly, it emphasizes the special role of P as the context for the pc function as well as a number of other functions. Therefore we shall use the dot notation '$P.$' for most of the functions which depend on partitionings and other objects. If it is clear from the context, which object is meant, we may omit this object, and just use the function name.

2.1 Length of Intervals in Partitions

It is very common to measure the length of intervals or the distance between time points in terms of time units. Examples are 'The train A arrives in the station 5 minutes before the train B leaves it'. 'Tomorrow I go on a adventure trip and will be back in 3 months time'.

A very useful function is therefore $P.length(t_1, t_2)$, which measures the length of the distance between t_1 and t_2 in terms of partitions of the partitioning P. For example, $month.length(t_1, t_2)$ measures the length of $[t_1, t_2[$ in months. Since partitions may have different lengths, this is a nontrivial operation.

The idea for the method can be illustrated with the following picture



The distance between t_1 and t_2 is the sum of the relative length of f_1, measured as a fraction of the length of partition 3, + the relative length of f_2, measured as a fraction of the length of partition 6, + the number of partitions in between.

Definition 2.5 (Length in Partitions) *Let P be a partitioning.*

For two time points t_1 and t_2 with $t_1 \leq t_2$ we define

$$P.lengthP(t_1, t_2) \stackrel{\text{def}}{=} \begin{cases} \dfrac{t_2 - t_1}{P.lopT(t_1)} & \text{if } P.pc(t_1) = P.pc(t_2) \\ P.pc(t_2) - P.pc(t_1) - 1 + & \\ \dfrac{P.eopT(t_1) - t_1}{P.lopT(t_1)} + \dfrac{t_2 - P.sopT(t_2)}{P.lopT(t_2)} & \text{otherwise} \end{cases}$$

If $t_2 < t_1$ then $P.lengthP(t_1, t_2) \stackrel{\text{def}}{=} -P.lengthP(t_2, t_1)$. ∎

$P.lengthP(t_1, t_2)$ is continuous. That means if t_1 is kept fixed and t_2 is moved, or the other way round, then $P.lengthP(t_1, t_2)$ makes no jumps. It is, however, not differentiable at the points where t_2 crosses the boundaries of neighbouring partitions with different length.

$P.lengthP(t_1, t_2)$ can be used to measure the absolute length of the interval $[t_1, t_2[$ if P is the partitioning for seconds or smaller time units. If P is the partitioning for minutes we can get the effect that an interval of 60 seconds length is smaller than one minute. This is the case for those minutes which contain leap seconds. Similar things happen for the coarser time units. We may get $day.length(t_1, t_2) < 1$ even if $hour.length(t_1, t_2) = 24$. This happens when daylight savings time is disabled just during the interval $[t_1, t_2[$ and the day is 25 hours long.

Definition 2.6 (modulo, remainder, $\lfloor ... \rfloor$ and $|\ldots|$)
The mod and remainder functions are used to map integers to non-negative indices $0, \ldots, n-1$. Therefore we need versions where the resulting values are between 0 and $n-1$, even for negative numbers. mod and remainder are defined for positive numbers as usual. For the negative numbers there are two different possibilities. We need the version where the resulting value is positive.

That means, k mod n is chosen such that for example 4 mod $3 = 1$ and -4 mod $3 = 2$.

$m/n = k$ remainder l is chosen such that for example $4/3 = 1$ remainder 1 and $-4/3 = -2$ remainder 2.

Let $\lfloor m \rfloor$ be the integer part of m such that $\lfloor 3.5 \rfloor = 3$ and $\lfloor -3.5 \rfloor = -3$.

For an interval s let $|s|$ be the length of the interval. ∎

2.2 Labels

For many periodic temporal notions there are standard names for the partitions. For example, days are named 'Monday', 'Tuesday' etc., months are named 'January', 'February' etc., seasons are named 'winter', 'spring' etc. If these names are attached as *labels* at the partitions, temporal notions like 'next summer' etc. can be modelled in an elegant way.

Definition 2.7 (Labelled Partitionings) *A* Labelling L *is a finite sequence of labels (strings)* $l_0, \ldots, l_{n-1}$.

A labelling $L = l_0, \ldots, l_{n-1}$ *is turned into a* labelling function $L(i)$ *for a coordinate* i*:*

$$L(i) \stackrel{\text{def}}{=} l_{i \bmod n}$$

In the sequel we shall identify the labelling L *with the labelling function* $L(i)$. ∎

A labelling L can now be very easily attached to a partitioning: the partition with coordinate i gets label $L(i)$.

Example 2.8 (The Labelling of Days) *The origin of the reference time is again January* 1^{st} *1970. This was a Thursday. Therefore we choose as labelling for the day partitioning*

$$L \stackrel{\text{def}}{=} Th, Fr, Sa, Su, Mo, Tu, We.$$

The following correspondences are obtained:

$ref.time:$	$\ldots$	$[-86400, 0[$	$[0, 86400[$	$[86400, 172800[$	$\ldots$
$coordinate:$	$\ldots$	-1	0	1	$\ldots$
$label:$	$\ldots$	We	Th	Fr	$\ldots$

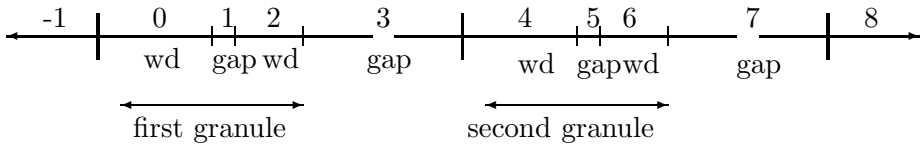
This means, for example, $L(-1) = We$*, i.e. December 31 1969 was a Wednesday.* ∎

2.3 Granules

The label 'gap' is a reserved keyword. It can be used to denote gaps between semantically related partitions. For example, if the partitions represent school holidays then the periods between the school holidays can be named 'gap'. Gaps, together with the possibility to use the same label at different positions in a labelling makes it possible to define *granules*, with the same effect as in the original definitions of granularities. As an example, consider again the definition of 'working day' as a period of time between 8 o'clock am and 5 o'clock pm, interrupted by a lunch break between 1 o'clock pm and 2 o'clock pm. This could be defined in two stages. First, we define a partitioning with an anchor time at some particular Monday 8 o'clock am, and durations '5 hour', '1 hour', '3 hour', '16 hour'. The '1 hour' period is the lunch break and the '16 hour' period is the time between two 'working days'. A suitable labelling is 'working_day, gap, working_day, gap'. A *granule* is characterized as a maximally long subsequence of a labelling such that the non-gap labels in the granule are the same. 'working_day, gap, working_day' in the above labelling is a granule, and the only granule.

Remark 2.9 *The definition of a* granule *in this paper includes internal gaps. A granule therefore corresponds always to a* convex *interval. Since a granule is generated by a labelling, the information about which part of the granule is a gap, is always available. Therefore it is technically simpler to define it this way, and to let the algorithms exploit information about internal gaps.* ∎

If a labelling is attached at a partitioning, we get granules as subsets of the time line. The next picture illustrates this for the 'working_day' example. 'wd' stands for 'working_day'.



Only the partitions 0-2, 4-6 etc. represent granules. The partitions 2-4 are not a granule, although they are also labelled 'working_day, gap, working_day'. This is prevented because the definition of granule is based on the initial labelling, in this case 'working_day, gap, working_day, gap'.

Definition 2.10 (Granule) *Let $L = l_0, \ldots, l_{n-1}$ be a labelling.*

1. *A* granule of the labelling L *is a maximal subsequence $G = l_k \ldots l_m$ of L such that*
   (i) $l_k \neq gap$ and $l_m \neq gap$, and
   (ii) all non-gap labels in G are the same.

2. *A granule of a labelling function $L(\ldots)$ (Def. 2.7) is a pair (k, m) of coordinates, such that $l_{k \bmod n}, \ldots l_{m \bmod n}$ is a granule of the labelling L.*

■

A labelling like 'Monday, Tuesday, ...' without gaps and with the labels all being different has granules consisting of a single label each. A labelling may of course also have several non-trivial granules. To see this, let us take our 'working_day' example a bit further. The underlying partitioning partitions also the weekends. If we want to specify that 'working_days' are only from Monday till Friday, we could do this with the following labelling: 'wdMo, gap, wdMo, gap, wdTu, gap, wdTu, gap, wdWe, gap, wdWe, gap, wdTh, gap, wdTh, gap, wdFr, gap, wdFr, gap, gap, gap, gap, gap, gap, gap, gap, gap'. This labelling has five different granules 'wdMo, gap, wdMo', 'wdTu, gap, wdTu', 'wdWe, gap, wdWe', 'wdTh, gap, wdTh', 'wdFr, gap, wdFr', one for each day. The last 8 gaps exclude the weekend.

Each partition in a labelled granule can either be part of a granule or not be part of a granule. If it is not part of a granule, then it is gap-labelled, but the neighbour non-gap labels are different.

A partitioning without an explicit labelling defines of course also granules: each partition is a granule consisting just of this single partition.

We define a function *closestGranule* which returns the coordinates of the closest granule. It comes in two versions. The version $closestGranuleC(i)$ takes a coordinate as argument, and the version $closestGranuleT(t)$ takes a time point as argument.

Definition 2.11 (Closest Granule) *Let L be a labelling of a partitioning P, i a coordinate and t a time point.*

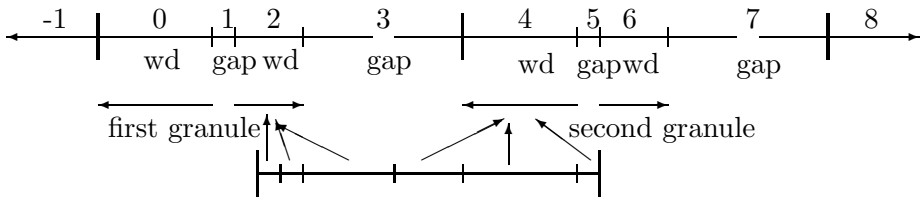
The function $L.closestGranuleC(i)$ returns a pair (k, m) of coordinates, such that
(i) (k, m) is a granule of the labelling function $L(\ldots)$ (Def. 2.10), and
(ii) either i is within a granule, i.e. $k \leq i \leq m$, or i is not within a granule, but it is closer to the granule (k, m) than to any other granule. That means precisely, if (a, b) are the boundaries of another granule, and $b \leq i \leq k$ then $k - i \leq i - b$, and if $m \leq i \leq a$ then $a - i \leq i - m$.

The function $L.closestGranuleT(t)$ returns a pair (k, m) of coordinates, such that
(i) (k, m) is a granule of the labelling function $L(\ldots)$, and
(ii) either t is within a granule, t is not within a granule, but it is closer to the granule (k, m) than to any other granule. That means precisely, if (a, b) are the boundaries of another granule, and $P.eopC(b) \leq t \leq P.sopC(k)$ then $P.sopC(k) - t \leq t - P.eopC(b)$, and if $P.eopC(m) \leq t \leq P.sopC(a)$ then $P.sopC(a) - t \leq t - P.eopC(m)$. ■

Notice that *closestGranule* prefers granules which lie in the future of i or t. That means, if i or t is exactly in the middle between two granules then the future one is chosen.

Length of intervals in granules:
Once the notion of 'working_day' is defined by means of granules, it is quite natural to measure intervals in terms of the length of a 'working_day'. For example, you may want to say that 'these four hours are half a working day'. This is not much of a problem if the granules all have the same length. If not, it depends on the position of the interval. 'Half a working day' may mean something different if I measure it on a Monday where I work 8 hours, or on a Friday, where I work, say, only 4 hours. Even worse, what if the interval lies on a weekend, and therefore outside any granules? It may still make sense to say that 'these four hours are half a working day', if I consider to shift the interval into a working day.

The function $lengthG(t_1, t_2)$ measures the distance between t_1 and t_2 in terms of granules. It deals with the problem that part of the interval $[t_1, t_2[$ or even the whole interval may lie outside any granule. The basic idea is to split the interval $[t_1, t_2[$ according to the given partitioning, determine for each subinterval the closest granule (Def. 2.11), and measure the subinterval in terms of this closest granule. The arrows in the next picture show which granules are used to measure the different parts of the interval.



Definition 2.12 (Length in Granules) *Let P be a partitioning which is labelled with a labelling L. The function $P.lengthG(t_1, t_2)$ measures the distance between t_1 and t_2 in terms of granules as follows:*

Step 1: The interval $[t_1, t_2[$ is partitioned into subintervals $s_1, \ldots$ such that the subinterval boundaries are aligned with the partition boundaries of P.

Step 2: The middle point t_i for each subinterval s_i is computed, and the coordinates (l, m) of the granule closest to t_i is determined by the closestGranuleT function (Def. 2.11). Let l_i be the length of the non-gap partitions in this granule.

Step 3: The relative lengths $|s_i|/l_i$ are added together to give the result of $P.lengthG(t_1, t_2)$.

■

The *lengthG* function yields intuitively correct results if the interval $[t_1, t_2[$ is a subset of the gap or non-gap partitions of a granule. For intervals lying outside any granule, it is questionable whether the closest granule is the right reference granule. It may always be the next future granule, or it may be a very particular reference granule. These options are not built-in in PartLib, but they can easily be programmed using its interface functions.

2.4 Relations Between Partitionings

PartLib supports four different relations between partitionings.

Definition 2.13 (Relations Between Partitionings) *Let P and Q be two partitionings. We define the following four relations between P and Q.*

1. *P has shorter partitions than Q if the largest partition of P is shorter than the shortest partition of Q (P is finer grained than Q.)*

2. *P has shorter granules than Q if the largest granule of P is shorter than the shortest granule of Q.*

3. *P includes the partitions of Q if each partition of P is a subset of a partition of Q.*

4. *P includes the granules of Q if each granule of P is a subset of a granule of Q.* ∎

These four relations are easy to define, but difficult to compute because for algorithmic partitionings it is not possible to compute the minimal or maximal partition length. Therefore PartLib uses an approximation. It generates random time points and computes the partition and granule length for these random points and certain points in their neighbourhood. How many points in the neighbourhood are to be checked is controlled by the 'repetition' parameter. For example, for the partitioning 'month' one would check 12 subsequent months for each random time point. This way, the 'local structure' of the partitionings is checked completely, whereas only finite samples check the 'global structure', in the 'month' example, the leap years.

The relations 'includes the partitions of' and 'includes the granules of' are also checked with finitely many randomly generated time points and their neighbourhoods.

3 Date Formats

In the subsequent section various notions are introduced in a recursive way: date formats, shifts, durations, and different types of partitionings.

We start with the definition of date formats.

Definition 3.1 (Date Format) *A* date format *DF is a sequence $P_0/\ldots/P_k$ of partitionings.*
A date *in a date format DF is a sequence $d_0/\ldots/d_n$ of integers with $n \leq k$.* ∎

In principle, the date formats can consist of arbitrary partitionings. In most calendar systems there are, however, a few particular date formats. The Gregorian calendar, for example, has the two date formats year/month/day/hour/minute/second (where the names stand for the corresponding partitions), and year/week/day/hour/minute/second.

There are, however, two big differences between common date formats and the particular interpretation of the numbers in the dates we need in this paper. Consider the date format year/month/day/hour/minute/second. The first difference is the interpretation of the year. The number 30, for example, for the 'year' part in the date format is the coordinate of the year. If the year 1970 has coordinate 0 then '30' is the coordinate of the year 2000. The next difference has to do with the way we count months, weeks and days. Usually these are counted from 1. That means, January is month 1, the first week in a year is week 1, and Monday is day 1. In contrast to this, we count hours, minutes and seconds from 0. The first hour in a day is hour 0, the first minute in an hour is minute 0 etc. Our interpretation of date formats like the above is that months, days etc. denote shifts, instead of absolute values. In this interpretation the date 30/1 denotes a shift of 1 month from the beginning of the year with coordinate 30 (i.e. the year 2000). This is the beginning of February. Thus, all time units are counted from 0.

By interpreting the numbers as shifts, there is no problem to deal with arbitrary big numbers, and even with negative numbers. The date 30/200/-50, for example, in the date format year/month/day/hour/minute/second denotes a time point t which is obtained from the beginning s of the year 2000 (30 years after 1970), by shifting s 200 months into the future, and from there 50 days into the past.

With these ideas in mind we can define a function *date*, which turns a time point into a date of the given format.

Definition 3.2 (*date*) *Let $DF = P_0/\ldots/P_k$ be a date format, and t a time point.*
Let $d_0/\ldots/d_k \stackrel{\text{def}}{=} DF.date(t)$ where
$d_0 \stackrel{\text{def}}{=} P_0.pc(t)$ *and*
$t_i \stackrel{\text{def}}{=} P_i.sopT(P_{i-1}.sopT(t))$ *and*
$d_i \stackrel{\text{def}}{=} \lfloor P_i.lengthP(t_i, t) \rfloor$ *for* $i = 1 \ldots k$. ∎

d_0 is the absolute coordinate of the P_0 partition containing t. t_1 is the starting point of the P_1 partition containing t. The d_i are then calculated as P_i increments compared to t_i where t_i is the beginning of the p_i partition containing t.

For example, in the date format year/week/day/... d_0 is the coordinate of the year containing t. t_1 is the beginning of the first week overlapping with the year d_0. d_1 is then the integer part of the number of weeks between t_1 and t. t_2 is the beginning of the first day in the week containing t etc.

The *date* function is exact only if the last partitioning P_k corresponds to the integers of the time axis, usually seconds, or fractions of a second.

It is also possible to turn a date $d = d_0/\ldots/d_n$ in a date format $DF = P_0/\ldots/P_k$ into a time point. The function $TimePoint(d)$ gives the dates a precise semantics.

Definition 3.3 (Time Point $DF.TimePoint(d)$) *Let $DF = P_0/\ldots/P_k$ be a date format and let $d = d_0/\ldots/d_n$ be a date in this format. The corresponding time point is defined:*

$DF.TimePoint(d) \stackrel{\text{def}}{=} t_n$ *where*

$t_0 \stackrel{\text{def}}{=} P_0.sopC(d_0)$ *and*

$t_i \stackrel{\text{def}}{=} P_i.sopC(P_i.pc(t_{i-1}) + d_i)$ *for* $i = 1 \ldots n$. ∎

In the date format year/week/day/..., for example, t_0 is the beginning of the year with coordinate d_0. $week.pc(t_0)$ is the coordinate of the week containing t_0. $week.pc(t_0) + d_1$ is the coordinate of the week which is d_1 weeks into the year. $t_1 = week.sopC(week.pc(t_0) + d_1)$ is the beginning of this week. t_2 is then the beginning of the day which is d_2 days into this week etc. Finally, t_k is the coordinate of the second denoted by the given date.

By induction on the length of a date, one can prove that turning a time point t into a date and the date back into a time point then we end up at the same time point t.

Proposition 3.4 *For a date format DF whose last partitioning corresponds to the integers in the time axis, and a time point t:*

$$DF.TimePoint(DF.date(t)) = t.$$

∎

The other direction, $DF.date(DF.TimePoint(d)) = d.$ need not hold, because there may be quite different dates which represent the same time point. For example, February 1st 2000 may be represented by 30/1/0 or by 30/0/31.

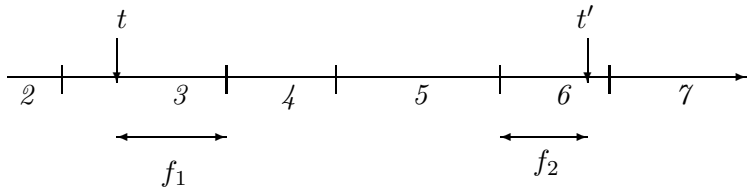
4 Shift Functions

Notions like 'in two weeks time' or 'three years from now' etc. denote time shifts. They can be realized by a function which maps a time point t to a time point t' such that $t' - t$ is just the required distance of 'two weeks' or 'three years' etc. Shift functions are of particular importance to the PartLib library because some of the specification mechanisms for partitionings require to shift an anchor point by certain durations. Therefore this needs to be explained in detail.

4.1 Length Oriented Shift Function

We define a function $P.shiftPL(t, m)$ which shifts a time point t to a time point $t' = P.shiftPL(t, m)$ such that $P.lengthP(t' - t) = m$. ('shiftPL' stands for 'shift Partitions Length oriented', in contrast to 'shiftPD', which stands for 'shift Partitions Date oriented').

Example 4.1 (for shiftPL) *The algorithm for this function can be best understood by the following example:*



Suppose we want to shift the time point t by 3.5 partitions. First, the relative distance f_1 between t and the end of the partition containing t is measured. Suppose it is 0.75. That means from the end of the partition we need to move forward still 2.75 partitions. We can move forward 2 partitions by just adding the 2 to the coordinate 4. We end up at the start of partition 6. From there we need to move forward 0.75 partitions, which is just 75% of the length of partition 6. ■

Definition 4.2 (shiftPL) *Let P be a partitioning, t a time point, and m a real number.*
If $m \geq 0$ then

$$shiftPL(t) \stackrel{\text{def}}{=} \begin{cases} t + m * lopT(t) & \text{if } t + m \cdot lopT(t) \leq eopT(t) \\ sopC(pc(t) + \lfloor m \rfloor) + (m - \lfloor m \rfloor) \cdot lopC(pc(t) + \lfloor m \rfloor) & \text{if } sopT(t) = t \\ sopC(pc(t) + \lfloor m' \rfloor + 1) & \\ \quad + (m - m') \cdot lopC(pc(t) + \lfloor m' \rfloor + 1) & \text{otherwise} \end{cases}$$

where $m' \stackrel{\text{def}}{=} m - (eopT(t) - t)/lopT(t)$.
If $m < 0$ then

$$shiftPL(t) \stackrel{\text{def}}{=} \begin{cases} t + m * lopT(t) & \text{if } sopT(t) \leq t + m \cdot lopT(t) \\ sopC(pc(t) + \lfloor m \rfloor) + (m - \lfloor m \rfloor) \cdot lopC(pc(t) + \lfloor m \rfloor - 1) & \text{if } sopT(t) = t \\ sopC(pc(t) + \lfloor m' \rfloor) & \\ \quad +(m - m') \cdot lopC(pc(t) + \lfloor m' \rfloor - 1) & \text{otherwise} \end{cases}$$

where $m' \stackrel{\text{def}}{=} m + (t - sopT(t))/lopT(t)$.

■

It is not so difficult to see that the shiftPL function shifts a time point t by m partitions to a time point t' such that the distance $t' - t$ is just m.

Proposition 4.3 (Soundness of shiftPL) *For any time point t:*

$$shiftPL(t, m) - t = lengthP(t, shiftPL(t, m)) = m$$

Furthermore

$$shiftPL(shiftPL(t, m_1), m_2) = shiftPL(t, m_1 + m_2)$$

■

The proof is technical and gives no new insight. It is therefore omitted.

Unfortunately the shiftPL function does not always give intuitive results. Suppose the time point t is noon at March, 15th, and we want to shift t by 1 month. March has 31 days. Therefore the distance to the end of March is exactly 0.5 months. Thus, we need to move exactly 0.5 times the length of April into April. April has 30 days. 0.5 times its length is exactly 14 days. Thus, we end up at midnight April, 14th.

This is not what one would usually expect. We would expect to shift t to the same time of the day as we started with. With the above definition of shiftPL this happens only by chance, or when the partitions have the same length.

4.2 Date Oriented Shift Function

PartLib provides another shift function, $shiftPD$ which avoids the above problems and gives more intuitive results. The idea is to do the calculations not on the level of reference time points, but on the level of dates. If, for example, t represents 2004/1/15, then 'in one month time' usually means 2004/2/15. That means the reference time must be turned into a date, the date must be manipulated, and then the manipulated date is turned back into a reference time. This is quite straight forward if the partitioning represents a basic time unit of a calendar system

(year, month, week, day etc.), and this calendar system has a date format where the time unit occurs. In the Gregorian calendar this is the case, even for the time unit 'weeks'. 'In two weeks time' requires to turn the reference time into a date format which uses weeks. The corresponding date format uses the counting of weeks in the year (ISO 8601). For example, 2004/42/1 means Tuesday[1] in week 42 in the year 2004. In two weeks time would then be 2004/45/1.

The next problem is to deal with fractional shifts. How can one implement, say, 'in 3.5 months time'? The idea is as follows: suppose the date format is year/month/day/hour/minute/second, and the reference time corresponds to, say, 34/1/20/10/5/1. First we make a shift by three months and we end up at 34/4/20/10/5/1. This is a day in May. From the date format we take the information that the next finer grained time unit is 'day'. May has 31 days. $0.5 * 31 = 15.5$. Therefore we need to shift the date first by 15 days, and we end up at 34/4/34/10/5/1. There is still a remaining shift of half a day. The next finer grained time unit is hour. One day has 24 hours. $0.5 * 24 = 12$. Thus, the last date is shifted by 12 hours, and the final date is now 34/4/34/22/5/1. This is turned back into a reference time.

This version of 'shift' gives more intuitive results. The drawback is that $shiftPD(t, m) - t = lengthP(t, shiftPD(t, m)) = m$ is usually no longer true. $shiftPD$ has in fact not much to do with $lengthP$.

The concrete definition of $shiftPD$ depends on the partitioning type. Therefore, we give them in the corresponding sections below (see Def. 8.5).

4.3 Shift by Granules

A statement like 'we must move this task by three working days' refers to a shift of time points which is measured in granules. Since granules are multiples of partitions, it is not necessary to have a $shiftGranules$ function. A simpler idea is to turn a number m of granules into a corresponding number n of partitions and then to apply the date oriented or length oriented shift function for partitions. Therefore PartLib provides only a function $granules2Partitions(t, m)$. It turns the number m of granules into a number n of partitions such that a shift of the time point t by m granules corresponds to a shift by n partitions. m can be any positive or negative fractional number.

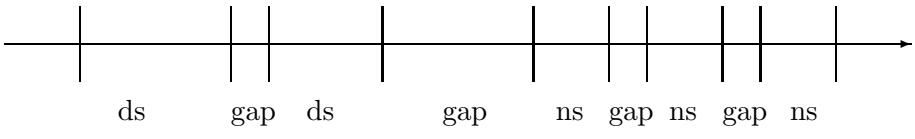
The algorithm for $granules2Partitions$ has to distinguish different cases:

case 1: the time point t is within a granule;

- **case 1a:** t is within a non-gap partition of the granule;
- **case 1b:** t is within a gap partition of the granule;

case 2: t is in a gap partition between two granules. This case can be reduced to case 1a by inverting the role of gaps and granules. That means the granules, with the intra–granule gaps, become gaps, and the gaps between the granules become granules.

These cases are quite trivial if the structure of the granules is always the same. An example where the granules have a different structure may be a working day consisting of a day shift (ds) and a night shift (ns). As indicated in the figure below the day shift consists of two non-gap blocks (morning and afternoon) separated by a lunch break. The night shift consists of three non-gap blocks separated by two breaks.

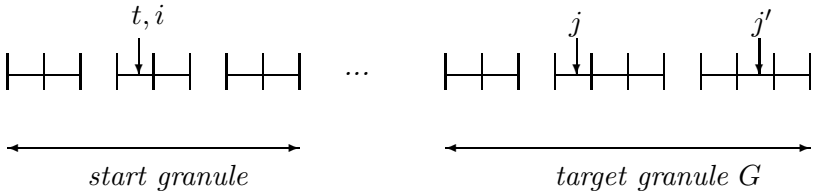


[1] According to ISO 8601, the first day in a week is Monday. In the standard notation this is day number 1. Since we count days from 0, Monday is day 0 and Tuesday is day 1.

Let us illustrate the difficulties with a still quite simple example. If the time point t is, say, in the afternoon block of the day shift, where should 'one granule further' be? It should be in the night shift, but not in one of the breaks of the night shift. The function *granules2Partitions* tries to identify in this case a corresponding non-gap block in the night shift by measuring the relative distance of the non-gap block containing t to the left boundary of the granule, in this case 2/2 (t is in non-gap block 2 of two non-gap blocks). Since the night shift contains three non-gap blocks, $2/2 * 3 = 3$, i.e. the corresponding non-gap block in the night shift is the third block. This is a heuristic. Experience has to show whether there are better ones.

We present the cases 1a and 1b of the algorithm for *granules2Partitions* by means of typical examples. Only the case $m > 0$ is explained in detail. The case $m < 0$ is analogous.

Example 4.4 (for case 1a of *granules2Partitions*:) *Suppose we want to shift the time point t in the picture below by $m = 1.5$ granules. t is within a non-gap partition of a granule and $m > 0$. We determine the number of partitions to be shifted in 5 steps.*



Let $i \stackrel{\text{def}}{=} P.pc(t)$ be the coordinate of the partition containing t. Let b be the relative distance of the non-gap block containing t. In the example t is in block 2 of three blocks. Therefore $b = 2/3$. Let d be the relative distance of partition i within the non-gap block containing t. In the example t is in partition 1 of a 2-partition non-gap block. Therefore $d = 1/2$.

Step 1: we determine the target granule G into which the time point is to be shifted by moving from the start granule containing t $\lfloor m \rfloor$ granules forward. In this example, it is just one granule further.

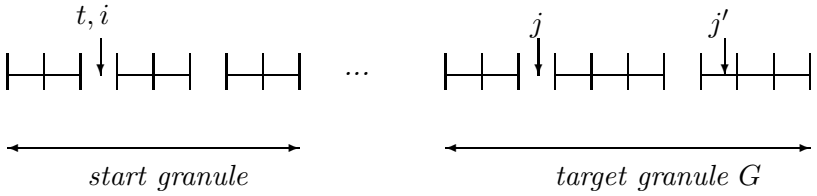
Step 2: we determine the target block B in the target granule G as $b' = b * b_G$ where b_G is the number of non-gap blocks in G. The target granule in the example has 3 non-gap blocks. Therefore $b' = 2/3 * 3 = 2$, i.e. the second block is the target non-gap block in G.

Step 3: we determine the target partition in block B by $d' = d * d_G$ where d_G is the number of non-gap partitions in the target block. In the example $d' = 1/2 * 3 = 1.5 \sim 2$, i.e. the second partition in block B is the target partition. Let j be the coordinate of this partition.

Step 4: Let $m \stackrel{\text{def}}{=} m - \lfloor m \rfloor$. If $m' = 0$ then $j - i$ is the result of granules2Partitions. If $m' > 0$ let $e \stackrel{\text{def}}{=} m' * |G|$ where $|G|$ is the number of non-gap partitions in G. In the example $e = 0.5 * 8 = 4$. Let j' be the partition obtained by moving from j e non-gap partitions forward. If j' is within the target granule G, then $j' - i$ is the result of granules2Partitions.

Step 5: If j' is beyond the target granule G, let f be the relative distance between the partition j and the end of the granule G, counting only non-gap partitions. In the example $f = 5/8$. Since j' is beyond the target granule G, $f > m'$. Let G' be the next granule after G. We must move the target partition into G'. Now let j'' be the coordinate of the partition obtained by moving $\lfloor (f - m') * |G'| \rfloor$ non-gap partitions forward from the start of granule G'. $j'' - i$ is now the result of granules2Partitions. ■

Example 4.5 (for case 1b of *granules2Partitions*:) *Suppose we want to shift the time point t in the picture below by $m = 1.5$ granules. t is within a gap partition of a granule and $m > 0$. We determine the number of partitions to be shifted in 5 steps.*



Let $i \stackrel{\text{def}}{=} P.pc(t)$ be the coordinate of the partition containing t. Let b be the relative distance of the gap block containing t. In the example t is in block 1 of two blocks. Therefore $b = 1/2$. Let d be the relative distance of partition i within the block containing t. In the example t is in partition 1 of a 1-partition gap block. Therefore $d = 1$.

Step 1: we determine the target granule G into which the time point is to be shifted by moving from the start granule containing t $\lfloor m \rfloor$ granules forward. In this example, it is just one granule further. If G contains no gaps at all, we proceed as in case 1a (Example 4.4), while ignoring that there are gaps in the start granule.

Step 2: we determine the target block B in the target granule G as $b' = b * b_G$ where b_G is the number of gap blocks in G. The target granule in the example has 2 non-gap blocks. Therefore $b' = 1/2 * 2 = 1$, i.e. block 1 is the target gap block in G.

Step 3: we determine the target partition in block B by $d' = d * d_G$ where d_G is the number of gap partitions in the target block. In the example $d' = 1 * 1 = 1$, i.e. the first partition in block B is the target partition. Let j' be the coordinate of this partition.

Step 4: Let $m' \stackrel{\text{def}}{=} m - \lfloor m \rfloor$. If $m' = 0$ then $j - i$ is the result of $granules2Partitions$. If $m' > 0$ we proceed as in case 1a, but ignore that there are gaps in the granules. Let $e \stackrel{\text{def}}{=} m' * |G|$ where $|G|$ is the number of partitions in G. In the example $e = 0.5 * 10 = 5$. Let j' be the partition obtained by moving from j e partitions forward. If j' is within the target granule G, then $j' - i$ is the result of $granules2Partitions$.

Step 5: If j' is beyond the target granule G, let f be the relative distance between the partition j and the end of the granule G, counting all partitions. In the example $f = 7/10$. Since j' is beyond the target granule G, $f > m'$. Let G' be the next granule after G. We must move the target partition into G'. Now let j'' be the coordinate of the partition obtained by moving $\lfloor (f - m') * |G'| \rfloor$ partitions forward from the start of granule G'. $j'' - i$ is now the result of $granules2Partitions$. ∎

5 Durations

The partitionings are the mathematical model of periodic time units, such as years, months etc. This offers the possibility to define *durations*. A duration may for example be '3 months + 2 weeks'. Months and weeks are represented as partitionings, and 3 and 2 denote the number of partitions in these partitionings. The numbers need not be integers, but can be arbitrary real numbers.

A duration can be interpreted as the length of an interval. The numbers should not be negative in this case. A duration, however, can also be interpreted as a time shift. In this interpretation negative numbers make perfectly sense. $d = (-2\ week), (-3\ month)$, for example, denotes a backward shift of 2 weeks followed by a backward shift of 3 months.

Definition 5.1 (Duration) *A* duration $d = (d_0, P_0), \ldots, (d_k, P_k)$ *is a list of pairs where the d_i are real numbers and the P_i are partitionings.*

If a duration is interpreted as a shift of a time point, it may be necessary to turn the shift around, in the backwards direction. Therefore the inverse of a duration is defined:
$-d \stackrel{\text{def}}{=} (-d_k, P_k), \ldots, (-d_0, P_0)$ ∎

For example, if $d = (3\ month), (2\ week)$ then $-d = (-2\ week), (-3\ month)$.

In Def. 8.5 below a `shift` function for algorithmic partitionings is introduced. It shifts a time point by a number of partitions in a given partitioning. Any such shift function can be lifted to operate on durations.

Definition 5.2 (`shift` for Durations) *Given a function $P.shift(t, m)$, which shifts a time point t by m partitions of the partitioning P, we define a corresponding* `shift` *function for durations:*

$$D.shift(t) \stackrel{\text{def}}{=} P_k.shift(\ldots P_1.shift(P_0.shift(t, d_0), d_1) \ldots, d_k)$$

where t is a time point and $D = (d_0, P_0), \ldots, (d_k, P_k)$ is a duration. ∎

For example, if $D = (3\ month), (2\ week)$ then

$$D.shift(t) = week.shift(month.shift(t, 3), 2),$$

i.e. t is shifted by 3 months first, and the resulting time point is then shifted by 2 weeks.

6 Calendar Systems

Calendar systems are essentially certain sets of partitionings. The Gregorian calendar, for example, can be modelled by defining suitable partitionings for years, months, weeks, days, hours, minutes and seconds. Since the CTTN–system has a global reference time, and there is always a mapping between the coordinate system of a partitioning and the global reference time, it is possible to navigate between different granularities of the same calendar system, and between different granularities of different calendar systems.

There is, however, more information associated with calendar systems. Date strings, for example, must be parsed in a calendar specific way. The fact, that we count months and weeks from number 1, but hours, minutes and seconds from number 0, must be encoded in a calendar specific parser for date strings.

If longer historical periods in a particular region of the world are to be covered, it may even be necessary to combine two or more calendar systems. In the western world it is the Julian and the Gregorian calendar system which have been used over the past 2000 years. Therefore the real calendar system is not either the Julian or the Greogorian system, but a combination of both.

By these and a few other reasons, 'calendar system' is not a concept in the PartLib library, but in a separate component of the CTTN–system. Therefore calendar systems are not further mentioned in this paper.

7 Global and Local Reference Time

The global reference time GRT corresponds directly to UTC time. With the introduction of a *local reference time* LRT for each partitioning it is possible to deal with leap seconds and time zones. The purpose is that the algorithms for the different partitions can just use the local reference time, and do not need to deal with leap seconds and time zones.

The transition from GRT to LRT is done in two steps. The first step deals with the leap seconds and the second step deals with time zones. The transition from LRT to GRT goes the other way round.

A correction function for leap seconds is defined first. The function $lsG(t)$ defined below ('G' for 'global') computes the accumulated leap seconds until the global reference time point t. The function $lsL(t)$ ('L' for 'local') also computes the accumulated leap seconds, but until the 'local' reference time point t. $lsG(t)$ is used for the transition from GRT to LRT, whereas $lsL(t)$ is used for the other direction. Unfortunately, it is computationally difficult to derive one version from the other. It is much more efficient when both functions are generated from a table of leap second corrections. (see http://www.ptb.de/de/org/4/43/432/ssec.htm).

Definition 7.1 (Correction Function for Leap Seconds) *If t is a time point in the global reference time then $lsG(t)$ computes the accumulated number of leap seconds until t.*

If t is a time point in a reference time where the leap second corrections have already been done then $lsL(t)$ computes the accumulated number of leap seconds until t. ∎

Example 7.2 (Correction Function for Leap Seconds) *The first 10 leap seconds were introduced for the last minute in 1971. The reference time for the regular end of this minute is 63072000. Therefore* $lsG(t) = 0$ *for all* $t \leq 63072000$ *and* $lsG(63072000+n) = n$ *for* $0 \leq n \leq 10$.

$lsG(t)$ *remains constant with value 10 from* $t = 63072010$ *until* $t = 94694410$. $lsG(94694411) = 11$*, because another leap second was introduced in the last minute of 1972.*

$lsL(t) = 0$ *for all* $t \leq 63072000$ *as well, but* $lsL(63072001) = 10$. $lsL(94694400) = 10$ *and* $lsL(94694401) = 11$ *etc.* ∎

Time zones are characterized by an offset between the local time and the time at the 0-meridian. For example, if at the 0-meridian it is 0 o'clock then the offset to the time zone of Germany is 1 hour, i.e. in Germany it is already 1 am. The time zone offset is in this case +3600 seconds, and the local reference time is 3600 seconds ahead of the global reference time.

Definition 7.3 (Transition between GRT and LRT) *Given the correction functions* lsG *and* lsL *for leap seconds (Def. 7.1) and a time zone offset* tzo*, we define*

$$LRT(t) \stackrel{\text{def}}{=} t + tzo - lsG(t).$$

for a global reference time t. $LRT(t)$ *computes the local reference time from the global reference time.*

The function

$$GRT(t) \stackrel{\text{def}}{=} t - tzo + lsL(t)$$

computes the global reference time from the local reference time. ∎

8 Specification of Partitionings

A partitioning is usually an infinite sequence of intervals, and these intervals need not be of the same length. Therefore it is not possible to specify such a partitioning by just enumerating its partitions.

Three basically different ways to specify partitionings are presented, and it is shown how a specification corresponds to a partitioning and an associated coordinate mapping. Since partitions themselves are not explicitly represented in PartLib, but only time points and coordinates, it is sufficient to give for each type of specification of a partitioning P corresponding definitions of the 'start of partition' function $sopC(i)$ and the 'partition coordinate' function $pc(t)$. $sopC(i)$ maps a coordinate i to the starting point of the corresponding partition and $pc(t)$ maps a time point t to the coordinate of the partition containing t. The two functions are fundamental for various service functions.

The 'start of partition' function determines the partitioning completely because

$$p_i \stackrel{\text{def}}{=} [sopC(i), sopC(i+1)[.$$

The 'partition coordinate' function $pc(t)$ is then derivable:

$$P.pc(t) \stackrel{\text{def}}{=} \min_i(P.sopC(i) \leq t < P.sopC(i)). \tag{1}$$

This way, however, $pc(t)$ can only be computed through search, which is extremely inefficient. Therefore we give a more efficient definition of $pc(t)$ in each case. In most cases the algorithm for $pc(t)$ tries at first a good guess i' for the coordinate, and then searches in the neighbourhood of i' until the condition (1) is satisfied.

8.1 Algorithmic Partitionings

The first type of partitionings is mainly used for modelling the basic time units of calendar systems, years, months etc. The specification consists of an average length of the partitions, a correction function and an offset against time point 0.

Definition 8.1 (Specification of Algorithmic Partitionings)
Algorithmic partitionings are specified by a the components (avl, po, cf, DF) *where*

1. *avl is the average length of a partition, given in the finest time unit;*
2. *po is an offset for the partition with coordinate 0, also given in the finest time unit,*
3. $cf(i)$ *is a correction function, and*
4. DF *is a date format. The date format is needed for the* $shiftPD$ *function.* ■

The correction function $cf(i)$ computes for a partition with coordinate i the difference between the reference time of the beginning of the partition with coordinate i, and the estimated beginning $i \cdot avl$.

Definition 8.2 (The Function *sopC* for Algorithmic Partitionings)
The algorithmic partitioning P *which is specified by the components* (avl, po, cf, DF) *(Def. 8.1) has the following 'start of partition' function:*

$$P.sopC(i) \stackrel{\text{def}}{=} GRT(i \cdot avl + cf(i) + po). \quad \blacksquare$$

Partitionings whose partitions have constant length in the local reference time only need a correction function that returns the constant 0. This is the case for seconds, minutes, and hours. It is no longer the case for days if daylight saving time regulations are taken into account.

Example 8.3 (Basic Time Units for the Gregorian Calendar)
The specification of the basic time units as algorithmic partitionings for the Gregorian Calendar are:

second: *average length: 1, offset: 0, correction function:* $\lambda(n)0$.

minute: *average length: 60, offset: 0, correction function:* $\lambda(n)0$.

hour: *average length: 3600, offset: 0, correction function:* $\lambda(n)0$.

day: *average length: 86400, offset: 0, correction function:* $-3600 \cdot h$ *if the day* i *is during the daylight saving time period, 0 otherwise.*
The number h *is usually 1 (for 1 hour). Exceptions are, for example, the year 1947 in Germany, where in the night of 1947/5/11 the clock was set forward a second time by 1 hour such that the offset against standard time was 2 hours.*

week: *average length: 604800, offset -259200, correction function: again, this function has to return an offset of* $-3600 \cdot h$ *for the weeks during the daylight saving time periods.*

month: *average length: 2592000 (30 days), offset 0, correction function: this function has to deal with the different length of the months and the daylight saving time regulations.*

year: *average length: 31536000 (365 days), offset 0, correction function: this function has to deal with leap years only. The effects of daylight saving time regulations are averaged out over the year.* ■

The 'partition coordinate' function $pc(t)$ maps a reference time point t to the coordinate of the partition containing t. For algorithmic partitionings this function is more complicated than $sopC(i)$ because it needs to use the correction function $cf(i)$, which takes a coordinate as input, and this is the coordinate which is yet to be computed. Therefore the basic idea for the algorithm is to use a fixed point iteration which calls $sopC(i)$ for guessed coordinates until the resulting time point matches the given time point. The algorithm is described rather informally, but the key steps should become clear.

Definition 8.4 (The Function $pc(t)$ for Algorithmic Partitionings) *Let t be a local reference time point for the given partitioning $P = (avl, po, cf, DF)$. The algorithm for $pc(i)$ starts with a first guess $i \stackrel{\text{def}}{=} t/avl$ for the coordinate of the partition containing t. Since this guess is wrong in general, there is a first iteration which brings i closer to the correct solution:*

Starting with an initial value for i', a fixed point iteration is performed until i' falls under a certain threshold[2]: Let $r \stackrel{\text{def}}{=} sopC(i)$. If $r \geq t$ let $r' \stackrel{\text{def}}{=} sopC(i-1)$ and compute $i' \stackrel{\text{def}}{=} (r-t)/(r-r')$ to get a better estimate $i \stackrel{\text{def}}{=} i - i'$ for the correct coordinate. If $r < t$, i is increased in a similar way[3].

The second phase of the algorithm is simpler: the correct coordinate is searched by just decreasing or increasing i by 1, until $sopC(i) \leq t < sopC(i+1)$ holds. The result of the function $pc(t)$ is then the coordinate i for which this condition holds. ∎

During the first phase, the algorithm performs big jumps to get very close to the correct solution. In the second phase it does the fine tuning by searching in the neighbourhood of the coordinate which was computed in the first phase. This phase guarantees that the result is correct, i.e. it satisfies the condition (1) for $pc(t)$. The algorithm is very efficient even if the average length of the partitions is quite different to their individual length.

We can now define the date oriented shift function for algorithmic partitionings. The idea of it was already explained in Section 4.2.

Definition 8.5 (Date Oriented `shiftPD` for Algorithmic Partitionings)
The function $P.shiftPD(t, m)$ where t is a GRT time point, m is a real number and P is an algorithmic partitioning (sec. 8.1) with date format $DF = P_0, \ldots$ performs the following steps:

1. *Let $d = d_1/\ldots/d_k \stackrel{\text{def}}{=} DF.date(t)$ (Def. 3.2);*
2. *i, d and m are now modified destructively:*
   while($m \neq 0$ and $i \leq k$)
   (a) $d_i \stackrel{\text{def}}{=} d_i + \lfloor m \rfloor$
   (b) *if($i < k$)*
   $t \stackrel{\text{def}}{=} DF.TimePoint(d)$ *(Def. 3.3)*
   $m \stackrel{\text{def}}{=} (m - \lfloor m \rfloor) \cdot P_{i+1}.lengthP(P_i.sopT(t), P_i.eopT(t))$
   $i \stackrel{\text{def}}{=} i + 1$.
3. *the result of $P.shiftPD(t, m)$ is now $DF.TimePoint(d)$.* ∎

Although the shiftPD function gives intuitive results in most cases, it has a number of drawbacks. One of them was already mentioned: shiftPD has not much to do with the lengthP function. Measuring the shifted distance with the lengthP function does usually not give the expected results.

Another drawback is that shifting a time point t first by m_1 partitions, and then by m_2 partitions may not be the same as shifting t by $m_1 + m_2$ partitions. This holds only if m_1 and m_2 are integers. A counter example for the case that m_1 and m_2 are factional values is:

Example 8.6 (Counterexample) *Suppose we want to shift the time point 0 twice by 1.5 months. The date for 0 is 0/0/0. Since February 1970 has 28 days, a shift by 1.5 months ends up at 0/1/14. Another shift by 1 month yields 0/2/14. This is in March. March has 31 days. Therefore a shift by 0.5 months means a shift by 15.5 days. We end up at 0/2/29/12. This is different to the result of a direct shift by 3 months: 0/3/0.*

[2]The threshold in the implementation is 3. The initial value for i' can be any number greater than the threshold. $i' = 10$ is fine.

[3]This version of the fixed point iteration is slightly simplified. It can happen that i' oscillates around the correct i. If this happens, the iteration is immediately stopped.

Nevertheless, the shiftPD is usually preferable. A striking example which illustrates the difference between shiftPD and shiftPL is such a simple thing as a shift by 1 day. If it is 5 pm, a shift by 1 day should end up at 5 pm next day. This is the case with the shiftPD function, even if during the night, standard time has been changed to daylight savings time. In contrast, the shiftPL function yields a very odd result in this case.

Other periodic temporal notions which can be modelled by algorithmic partitionings are, for example, sunrises and sunsets, moon phases, the church year which starts with Easter, etc. The specification for the western version of Easter would be: average length: 31536000 (1 year), offset: 7516800 (1970/3/29) and a correction function which actually computes the precise date of Easter (see for example [9]). The specification of all these partitionings must be accompanied by an appropriate `shiftPD` function.

The specification of the algorithmic partitionings requires the correction function $cf(i)$, and this is a piece of code. Therefore algorithmic partitionings are usually hard coded in the application program. This is different for the remaining three partitioning types. They can be specified purely symbolically. Therefore one can read their specification from a file or a database at run time. This makes the system very flexible.

8.2 Duration Partitionings

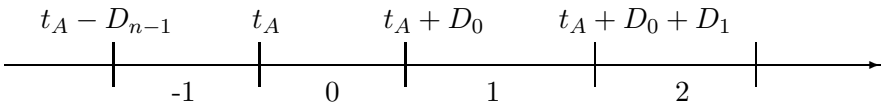
Duration partitionings are specified by an anchor time and a sequence of 'durations'. For example, I could define 'my weekend' as a *duration partitioning* with anchor time 2004/7/23, 4 pm (Friday July, 23rd, 2004, 4 pm) and durations: ('8 hour + 2 day', '4 day + 16 hour'). The first interval would be labelled 'weekend', and the second interval would be labelled 'gap'.

Definition 8.7 (Specification of a Duration Partitioning)
A duration partitioning is specified by the tuple $(t_A, (D_0 \dots D_{n-1}))$ *where*

1. t_A *is the anchor time point (in the global reference time),*
2. $D_0 \dots D_{n-1}$ *is a list of durations (Def. 5.1).*

■

The coordinates for a duration partitioning are such that the first partition after the anchor time point has coordinate 0. The next picture illustrates the situation.



The durations in the specification of a duration partitioning can be very irregular. Therefore there is not much of a chance to realize a 'start of partition' function *sopC* other than by just looping i times over $D_0 \dots D_{n-1}$.

Definition 8.8 (The Function *sopC* for Duration Partitionings)
The duration partitioning P *which is specified by the data* $(t_A, (D_0 \dots D_{n-1}))$ *(Def. 8.7) has the following 'start of partition' function:*

$$sopC(i) \stackrel{\text{def}}{=} t_i$$

where t_i *is determined by shifting the anchor time point* t_A i *times:*
Let $t_0 \stackrel{\text{def}}{=} t_A$.
if$(i \geq 0)$*: for* $j = 1, \dots, i$*:* $t_j \stackrel{\text{def}}{=} D_{(j-1) \bmod n}.shift(t_{j-1})$. *(Def. 5.2)*
if$(i < 0)$*: for* $j = 1, \dots, -i$*:* $t_j \stackrel{\text{def}}{=} -D_{(n-j) \bmod n}.shift(t_{j-1})$.

■

Notice that the shift function for durations uses the shiftPD function (Sec. 4.1). Because two shifts by m_1 and m_2 partitions are not necessarily the same as one shift by $m_1 + m_2$ partitions (Ex. 8.6) , this has an effect on the meaning of duration partitionings. A duration partitioning with a duration '1.5 month + 1.5 month' is not the same as a duration partitioning with a duration '3 month'.

Because there is no further assumption about the durations in the specification of duration partitionings, there is not much chance to optimize the 'partition coordinate' function *pc* either. Therefore the definition (1) is taken as algorithm for *pc*.

The `shiftPD` function cannot be optimized either. It also loops over the duration $D_0 \dots D_{n-1}$ and calls the shift function for durations as often as necessary.

Definition 8.9 (`shiftPD` for Duration Partitionings)
Let $P = (t_A, (D_0 \dots D_{n-1}))$ be a duration partitioning. The $P.shiftPD(t, m)$ function for a time point t and a real number m performs the following steps:

Let $(k$ remainder $l) \stackrel{\text{def}}{=} m/n$.

If $m \geq 0$:

if $(m \geq 1)$: for $i = 0, \dots, \lfloor m \rfloor - 1$ let $t \stackrel{\text{def}}{=} D_{i \bmod n}.shift(t)$
let $t \stackrel{\text{def}}{=} t + (m - \lfloor m \rfloor) \cdot (D_{\lfloor m \rfloor \bmod n}).shift(t) - t)$.

If $m < 0$:

for $i = 0, \dots, -\lfloor m \rfloor - 1$ let $t \stackrel{\text{def}}{=} -D_{n-1-(i \bmod n)}.shift(t)$
let $t \stackrel{\text{def}}{=} t + (m - \lfloor m \rfloor) \cdot (t - D_{(\lfloor m \rfloor \bmod n)}.shift(t)$.
The result of $P.shift(t, m)$ is now the modified t. ∎

8.3 Regular Partitionings

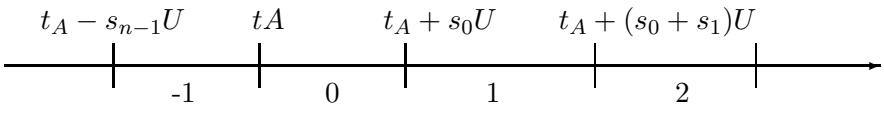
A special case of a duration partitioning is when all durations are of the form 'n P' and the partitioning P is the same in all durations. For this case there are more efficient ways than looping over lists of durations. Therefore, and because many partitionings are of this type, it makes sense to treat them in a special way.

A typical example is the notion of a semester at a university. In the Munich case, the dates could be: anchor time: October 2000. The shifts are: 6 months (with label 'winter semester') and 6 months (with label 'summer semester'). This defines a partitioning with partition 0 starting at the anchor time, and then extending into the past and the future. The partition with coordinate 0 in this example is the winter semester 2000/2001.

Definition 8.10 (Specification of Regular Partitionings) *A regular partitioning is specified by the triple $(t_A, U, (s_0 \dots s_{n-1}))$ where*

1. *t_A is the anchor time point (in the global reference time)*
2. *U is a partitioning, the time unit for the shifts,*
3. *$s_0 \dots s_{n-1}$ is a list of real numbers, the shifts.* ∎

The partitions of regular partitionings are obtained by shifting the anchor point t_A first by s_0 time units U, and then by $s_0 + s_1$ time units U etc.



This picture illustrates a subtle, but important difference to duration partitionings. The partition boundaries for duration partitions are computed by successively applying the shift function for the corresponding durations. The shift function for durations uses internally the date oriented shiftPD function, for which $shiftPD(shiftPD(t, m_1), m_2) \neq shiftPD(t, m_1 + m_2)$ may be the case.

In contrast to this the partition boundaries for regular partitions are computed by first adding the shifts together, and then shifting the anchor time in one single step. Both versions yield the same results for integer shifts. This is not guaranteed if the shifts are fractional.

Definition 8.11 () *The regular partitioning P which is specified by the data $(t_A, U, (s_0 \dots s_{n-1}))$ (Def. 8.10) has the following 'start of partition' function:*

$$sopC(i) \stackrel{\text{def}}{=} U.shiftPD(t_A, s(i))$$

where $s(i) \stackrel{\text{def}}{=} k \cdot \Sigma_{j=0}^{n-1} s_j + \begin{cases} \Sigma_{j=0}^{l} s_j & \text{if } i \geq 0 \\ \Sigma_{j=0}^{l-1} s_j & \text{otherwise} \end{cases}$

and $(k \text{ remainder } l) \stackrel{\text{def}}{=} i/n$.

■

Example 8.12 *Let the anchor date be 2000/1, and let the shifts be 3,4,5 months.*

$sopC(5)$ is calculated as follows:
$5/3 = 1$ *remainder* 2.
$i' = 1 \cdot (3 + 4 + 5) + (3 + 4) = 19$.
This means 2000/1 is to be shifted by 19 month, and we end up at the beginning of 2001/7.

$sopC(-5)$ is calculated as follows:
$-5/3 = -2$ *remainder* 1.
$i' = -2 \cdot (3 + 4 + 5) + 3 = -21$.
This means 2000/1 is to be shifted by -21 month, and we end up at the beginning of 1998/3. ■

The 'partition coordinate' function pc turns the reference time into a coordinate i of the time unit U and then uses the difference between i and the coordinate of the anchor time to compute the number of shifts which are necessary to get from the anchor time to the reference time. This is the coordinate of the partition containing t.

Definition 8.13 (The Function pc for Regular Partitionings)
Let $(t_A, U, (s_0 \dots s_{n-1}))$ be the specification of a regular partitioning. Let t be a local reference time point.
Let $i \stackrel{\text{def}}{=} U.pc(t) - U.pc(t_A)$ *and* $(k \text{ remainder } l) \stackrel{\text{def}}{=} i/\Sigma_{j=0}^{n-1} s_j$.

Let $P.pc(t) \stackrel{\text{def}}{=} k \cdot n + \max_{i \geq 0}((\Sigma_{j=0}^{i} s_j) \leq l)$. ■

Example 8.14 *Let the anchor date be 2000/1, and let the shifts be 3.5,4.5,5.5 months.*
Let t be such that $i \stackrel{\text{def}}{=} U.pc(t) - U.pc(t_A) = 21$.
k *remainder* $l = i/(3.5 + 4.5 + 5.5) = 1$ *remainder* 7.5.
$P.pc(t) = 1 \cdot 3 + 1 = 4$. *This is the partition between month 17 and 21.5.*

Now let t be such that $i \stackrel{\text{def}}{=} U.pc(t) - U.pc(t_A) = -21$.
k *remainder* $l = -21/13.5 = -2$ *remainder* 8.
$P.pc(t) = -2 \cdot 3 + 2 = -4$.
This is the partition between month -23.5 and -19. ■

shiftPD:
The `shift` function for regular partitionings is explained informally with the following example: Suppose we have a specification of *trimesters* in the following way: Anchor date: 2000/10, trimesters: 3,4,5 months.

We want to shift a time point t by 8.5 trimesters. The following steps are necessary

1. $8/3 = 2$ *remainder* 2. That means, first a shift of 2 full cycles of 3+4+5 = 12 months is performed, and we end up at a time point t_1.

2. The index i' of the partition containing t_1 is computed, suppose it is 1, i.e. t_1 lies in the first trimester. In order to get two trimesters further, a shift of $3 + 4 = 7$ months is necessary, and we end up at time point t_2 which is in the third trimester.

3. The third trimester has 5 month. $5 \cdot 0.5 = 2.5$, i.e. we shift t_2 by another 2.5 month.

Definition 8.15 (`shiftPD` for Regular Partitionings)
Let $P = (t_A, U, (s_0, \ldots, s_{n-1}))$ be a regular partitioning (Def. 8.10). The function $P.shiftPD(t, m)$ performs the following steps:

1. *Let $(k \text{ remainder } l) \stackrel{\text{def}}{=} \lfloor m \rfloor / n$*
   Let $t_1 \stackrel{\text{def}}{=} U.shiftPD(t, k \cdot \Sigma_{j=0}^{n-1} s_j)$.

2. *Let $i' \stackrel{\text{def}}{=} t_1^P$.*
   Let $t_2 \stackrel{\text{def}}{=} U.shiftPD(t, \Sigma_{j=0}^{l-1} s_{(i'+j) \bmod n})$.

3. *Return $U.shiftPD(t_2, (m - \lfloor m \rfloor) \cdot s_{(i'+l) \bmod n})$.*

■

Further examples for periodic temporal notions which can be encoded as regular partitionings are decades, centuries, the British financial year which starts April 1^{st}, the dates of a particular lecture, which is every week at the same time etc.

8.4 Date Partitionings

Date Partitionings are specified by providing the boundaries of the partitions as concrete dates.

An example could be the dates of the Time conferences: 1994/5/4 Time94 1994/5/5 gap 1995/4/26 Time95 1995/4/27 gap 1996/5/19 Time96 1996/5/21 gap 1997/5/10 Time97 1997/5/12 gap 1998/5/16 Time98 1998/5/18 gap 1999/5/1 Time99 1999/5/3 gap 2000/7/7 Time00 2000/7/10 gap 2001/6/14 Time01 2001/6/17 gap 2002/7/7 Time02 2002/7/10 gap 2003/7/8 Time03 2003/7/11 gap 2004/7/1 Time04 2004/7/4.

Another example could be the seasons: 2000/3/21 spring 2000/6/21 summer 2000/9/23 autumn 2000/12/21 winter 2001/3/21.

In the introduction I explained the trick how to turn these finitely many dates into an infinite partitioning: the differences between two subsequent dates are turned into durations. The durations are then used to extrapolate the partitioning into the infinity.

The seasons example above shows that this makes really sense. The time difference between the dates can be expressed as durations '3 month' for spring, '3 month + 2 day' for summer, '3 month - 2 day' for autumn and '3 month' for winter. The extrapolation of this now yields the seasons for the whole time line. This works even for the leap years, in which winter is one day longer. This is covered by the duration '3 month' for winter, which is one day longer in leap years.

Definition 8.16 (Specification of Date Partitionings) *A date partitioning is specified as a list dates $d_0, \ldots, d_n$ in a date format DF.* ■

The dates can very easily be turned into durations: Suppose the date format is $DF \stackrel{\text{def}}{=} P_0, \ldots$. The difference between two dates $d_i = d_{i,0}/d_{i,1}/\ldots$ and $d_{i+1} = d_{i+1,0}/d_{i+1,1}/\ldots$ yields the following duration: $(d_{i+1,0} - d_{i,0}, P_0)$, $(d_{i+1,1} - d_{i,1}, P_1), \ldots$, and this is sufficient to specify a duration partitioning. The anchor time is given by $DF.timePoint(d_0)$.

Notice that the coordinate calculation for duration partitionings which applies the shift function for durations, and this uses the date oriented shift function for partitionings, undos the above subtractions. Therefore it reconstructs exactly the original dates.

8.5 Folded Partitionings

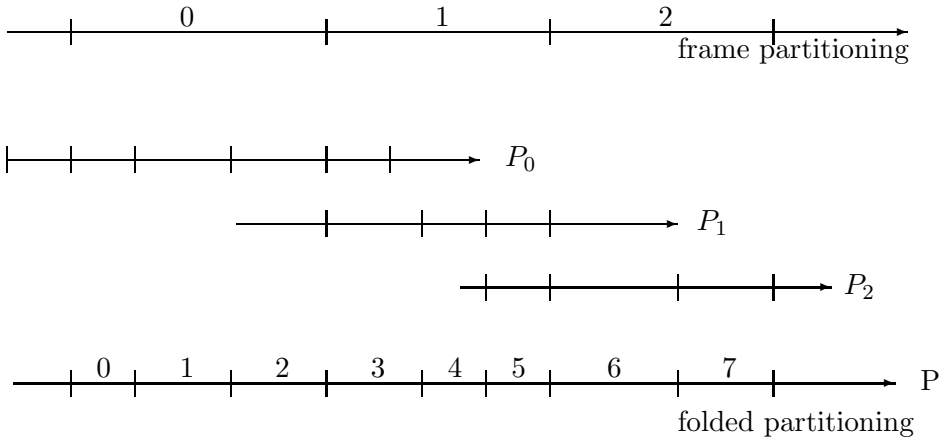
Another basic type of partitionings are *folded partitionings*. We explained already the bus timetable example. The bus timetable changes from season to season. The best way to specify this, would be to specify the seasons first, and for each season to specify the particular bus timetable. The 'folded partitioning' specification operation takes as input a *frame partitioning*, for example the seasons, and a sequence of *component partitionings*, for example the four different bus timetables. It maps the component partitionings automatically to the right frame partition, such that from the outside the whole thing looks like an ordinary partitioning.

Definition 8.17 (Folded Partitioning) *A folded partitioning P is specified by a* frame partitioning *F and a finite list $P_0, \ldots, P_{n-1}$ of* component partitionings*. The component partitionings must meet the following condition:*
The partitions of the component partitionings must be aligned with the partitions of the frame partition. That means, for $i = 0, \ldots, n-1$, the start of each frame partition must be the start of a component partition in P_i, and the end of each frame partition must be the end of a component partition in P_i;

A folded partitioning is called constant *iff each frame partition with coordinate i contains the same number of component partitions as the frame partition with coordinate i mod n.* ∎

The condition is in principle not necessary, but if it is not met, it complicates the algorithms enormously. It excludes, for example, that 'week' can be used as component partitioning when the frame partition is 'year'. 'month', however, can be used because years start and end with a month.

The figure below gives an idea how the coordinates for the folded partitioning are obtained from the coordinates of the frame partitioning and the component partitionings.



The algorithms for the 'partition coordinate' function *pc* and for the 'start of partition' function *sopC* are much more efficient if the partitioning is constant, i.e. the number of component partitions per frame partition of each component partitioning P_i does not change over time. In this case it is possible to compute the total number of component partitions up to the beginning of a given frame partition by a few multiplications and additions. If the partitioning is not constant, one must iterate through all frame partitions and add the corresponding numbers together.

The bus timetable example is typical for a folded partitioning which is *not* constant. This is due to the leap years. Since the winter is one day longer in a leap year, the bus timetable in this winter has more partitions than in the other winters. An example for a constant folded partitioning could be the definition of working hours with shifts. In the first week I may have a morning shift, in the second week an evening shift, and in the third week a night shift. The number of partitions labelled 'working hour' and 'gap' would not change in this case.

The 'partition coordinate' function pc needs an auxiliary function $partitionsUpTo$, which computes for a frame coordinate the number of partitions up to the beginning of this frame partition. More precisely: for a positive frame coordinate $frco$, $partitionsUpTo$ computes the number of component partitions from frame partition 0 (inclusive) up to the beginning of frame partition with coordinate $frco$, and for negative frame coordinates, it computes the *negated* number of component partitions from frame partition -1 (inclusive) down to and including the frame partition with coordinate $frco$.

We define this function for constant folded partitionings and for non-constant folded partitionings separately.

Definition 8.18 ($partitionsUpTo$ for constant folded partitionings) *Let P be a constant folded partitioning with component partitionings $P_0 \ldots, P_{n-1}$.*

We define the function $partitionsUpTo(frco)$ where $frco$ is a coordinate in the frame partitioning as follows:

For $i = 0, \ldots, n-1$ let $partitions(i)$ be the number of component partitions per frame partition P_i. Since the partitioning is constant, it is sufficient to compute these numbers for the first n frame partitions.

Let $cycle \stackrel{\text{def}}{=} \Sigma_{i=0}^{n-1} partitions(i)$.

Let $frco/n = blocks\ remainder\ rest$.

Now we have:

$partitionsUpTo(frco) \stackrel{\text{def}}{=} blocks * cycle + \Sigma_{i=0}^{rest-1} partitions(i)$. ∎

Definition 8.19 ($partitionsUpTo$ for normal folded partitionings) *Let P be a folded partitioning with frame partitioning F and component partitionings $P_0 \ldots, P_{n-1}$.*

We define the function $partitionsUpTo(frco)$ where $frco$ is a coordinate in the frame partitioning as follows:

If $frco \geq 0$ let $from \stackrel{\text{def}}{=} 0$ and $to \stackrel{\text{def}}{=} frco - 1$.
If $frco < 0$ let $from \stackrel{\text{def}}{=} frco + 1$ and $to \stackrel{\text{def}}{=} -1$.
Let $n \stackrel{\text{def}}{=} \Sigma_{i=from}^{to} P_{i \bmod n}.pc(F.eopC(i)) - P_{i \bmod n}.pc(F.sopC(i))$.
If $(frco \geq 0)$ let $partitionsUpTo(frco) \stackrel{\text{def}}{=} n$.
If $(frco < 0)$ let $partitionsUpTo(frco) \stackrel{\text{def}}{=} -n$. ∎

Now we can define the 'partition coordinate' function $pc(t)$. It uses the $partitionsUpTo$ function to get the number of partitions up to the frame partition containing t, and then computes the remaining partitions locally with the corresponding component partitioning.

Definition 8.20 (The Function pc for Folded Partitionings) *Let P be a folded partitioning with frame partitioning F and component partitionings $P_0, \ldots, P_{n-1}$. Let t be a global reference time point.*

Let $frco \stackrel{\text{def}}{=} F.pc(t)$ be the frame coordinate for time point t.

We have:

$pc(t) \stackrel{\text{def}}{=} partitionsUpTo(frco) + P_{frco \bmod n}.pc(t) - P_{frco \bmod n}.pc(F.sopC(frco))$ ∎

The 'start of partition' function $sopC$ needs an auxiliary function $frameCoordinate(co) = (frco, rist)$, which computes for a given coordinate of a partition (i) the coordinate of the frame partition which contains this partition and (ii) the coordinate of the first component partition within this frame partition. There are again different definitions for constant folded partitionings and for normal folded partitionings.

Definition 8.21 ($frameCoordinate$ for Constant Folded Partitionings) *Let P be a constant folded partitioning with component partitionings $P_0 \ldots, P_{n-1}$.*

We define the function $frameCoordinate(co)$ where co is a P-coordinate as follows:

For $i = 0, \ldots, n-1$ let $partitions(i)$ be the number of component partitions per frame partition P_i.

Let $cycle \stackrel{\text{def}}{=} \Sigma_{i=0}^{n-1} partitions(i)$.

Let $co/cycle = blocks\ remainder\ rest$

Let $frco \stackrel{\text{def}}{=} blocks \cdot n$ be the coordinate of the first frame partition p_0 in the partitions $p_0, \ldots, p_{n-1}$ which correspond to $P_0, \ldots, P_{n-1}$, and where one of the p_i contains the component partition with coordinate co.

Let $i \stackrel{\text{def}}{=} max_i(\Sigma_{k=0}^{i} partitions(k) \leq rest)$ be the offset from $frco$, such that $frco + i$ is the coordinate of the frame partition containing the component partition with coordinate co.

This way we get

$frameCoordinate(co) \stackrel{\text{def}}{=} (frco + i, blocks \cdot cycle + \Sigma_{k=0}^{i} partitions(k))$. ∎

Definition 8.22 ($frameCoordinate$ for Normal Folded Partitionings) *Let P be a constant folded partitioning with frame partitioning F and component partitionings $P_0 \ldots, P_{n-1}$.*

We define the function $frameCoordinate(co)$, where co is a P-coordinate as follows:

If $co \geq 0$ let

$frco \stackrel{\text{def}}{=} max_i(\Sigma_{k=0}^{i} P_{k\ mod\ n}.pc(F.eopC(k)) - P_{k\ mod\ n}.pc(F.sopC(k)) \leq co)$

$first \stackrel{\text{def}}{=} \Sigma_{k=0}^{frco} P_{k\ mod\ n}.pc(F.eopC(k)) - P_{k\ mod\ n}.pc(F.sopC(k)) \leq co)$

If $co < 0$ let

$frco \stackrel{\text{def}}{=} -max_i(\Sigma_{k=1}^{i} P_{-k\ mod\ n}.pc(F.eopC(-k)) - P_{-k\ mod\ n}.pc(F.sopC(-k)) \leq co)$

$first \stackrel{\text{def}}{=} \Sigma_{k=1}^{-frco} P_{-k\ mod\ n}.pc(F.eopC(-k)) - P_{-k\ mod\ n}.pc(F.sopC(-k)) \leq co)$

This way we get

$frameCoordinate(co) \stackrel{\text{def}}{=} (frco, first)$. ∎

The 'start of partition' function *sopC* is now defined as follows:

Definition 8.23 (The Function $sopC$ for Folded Partitionings)
Let P be a folded partitioning with frame partitioning F and component partitionings $P_0, \ldots, P_{n-1}$. Let co be a coordinate of P.

Let $(frco, first) \stackrel{\text{def}}{=} frameCoordinate(co)$

Let $border = F.sopC(frco)$ be the left boundary of the frame partition containing the component partition with coordinate co.

Now we define

$sopC(co) \stackrel{\text{def}}{=} P_{frco\ mod\ n}.sopC(P_{frco\ mod\ n}.pc(border) + (co - first))$ ∎

All operations on folded partitionings are straightforward if the coordinates involved remain within a single frame partition. The corresponding operation on the corresponding component partition can be used in this case. If the coordinates, however, cross the border of a frame partition then a special treatment is necessary. We explain this for the $shiftPD(t, m)$ function. All other functions are similar.

The $shiftPD(t, m)$ function for folded partitionings is straightforward if the shifted time point still remains in the same frame partition. In this case it is sufficient to use the $shiftPD$ function of the corresponding component partitioning. If this shift crosses the border of the frame partition then the relative distance to this border is subtracted from m, and then $shiftPD$ is called recursively for the border time point and the reduced m-value.

Definition 8.24 (`shiftPD` for Folded Partitionings) *Let P be a folded partitioning with frame partitioning F and component partitionings $P_0, \ldots, P_{n-1}$. Let t be a time point and m a real number.*

$P.shiftPD(t, m)$ performs the following steps:

1. *Let $co \stackrel{\text{def}}{=} F.pc(t)$ be the coordinate of the frame partition containing t.*
2. *Let $C \stackrel{\text{def}}{=} P_{co\ mod\ n}$ the component partitioning assigned to the partition containing t.*

3. *Let* $s \stackrel{\text{def}}{=} C.shiftPD(t, m)$ *be the shifted time point.*

4. *If* s *is still in the same frame partition as* t*, return* s*.*

5. *If* s *is not the in the same frame partition, let* m' *be the relative distance between* t *and the end* t' *(if* $m > 0$*) or the start* t' *(if* $m < 0$*) of the frame partition containing* t*.*

6. *Call* $P.shiftPD(t', m - m')$ *recursively.*

■

9 Service Functions

This section is for extra interface functions of PartLib. So far, there is just one of them, the 'which' function.

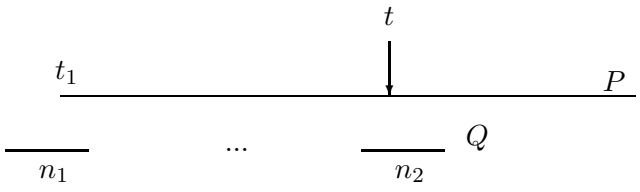
9.1 Which

The 'which' function can be used to answer queries like 'which day in the week is now' or 'in which week in the year is the time point t'. The parameters are $which(t, P, Q, inclusion, asGranule)$. t is the time point for which we want to get the information. P and Q are partitionings. For example, P could be the 'week' partitioning and Q could be the 'year' partitioning. *inclusion* is a control parameter for determining what counts as the first P partition in the Q partitioning. The problem can be best illustrated with weeks and years. Since weeks and years are not synchronized, the beginning of a year need not be the beginning of a week. It is therefore a matter of convention what counts as the first week in a year. It could be the first week which is completely contained in the year, it could be the first week which overlaps with the year, or it could be the first week whose lager part is in the year. The latter one is the condition which is commonly being used. Therefore *inclusion* is one of the keywords 'subset', 'overlap' and 'bigger_part_inside'.

The last parameter, *asGranule* controls whether partitions or granules are counted.

The 'which' function is a partial function. There are two conditions which can cause $which(t, P, Q, inclusion, asGranule)$ to be undefined. The first one is that the Q-partition is too small for a suitable P-partition. The second one is that $asGranule = true$ and the time point t is not within a granule. Then it is not clear how to count, and therefore the function shoud be undefined. Nevertheless the actual implementation returns in this case the result for the the smallest $t' > t$, for which this condition is not met. An extra result flag indicates that this has happened.

'which' first locates the Q-partition containing t. Then it determines the first P-partition in this Q-partition. Finally it counts the number of P-partitions which must be traversed until t is reached. The following figure illustrates the situation.



Definition 9.1 (which) *We define a function* $which(t, P, Q, inclusion, asGranule)$ *where* t *is a time point,* P *and* Q *are partitioninings, inclusion is one of the keywords, 'subset', 'overlap' or 'bigger_part_inside', and asGranule is a Boolean flag.*
Let $t_1 \stackrel{\text{def}}{=} Q.sopT(t)$ *be the start point of the* Q*-partition containing* t *and*
let $t_2 \stackrel{\text{def}}{=} Q.eopT(t)$ *be the end point of the* Q*-partition containing* t*.*

Let $n_1 \stackrel{\text{def}}{=} P.pc(t_1)$ *be the* P*-partition containing* t_1*.*
Let $n_2 \stackrel{\text{def}}{=} P.pc(t)$ *be the* P*-coordinate of* t*.*

Case: $asGranule = false$
If $inclusion = subset$ *and* $P.sopC(n_1) < t_1$*, increase* n_1 *by 1.*
If $P.eopC(n_1) > t2$ *then 'which' is undefined.*

If $inclusion = bigger\_part\_inside$ *and* $t_1 - P.sopC(n_1) > P.eopC(n_1) - t_1$*, increase* n_1 *by 1.*
If $P.eopC(n_1) > t2$ *then 'which' is undefined.*

Now let $which(\ldots) \stackrel{\text{def}}{=} n2 - n1$*.*

Case: $asGranule = true$

Let (c_1, c_2) *be the start and end coordinates of the first granule which contains* t_1*, or, if* t_1 *is between granules, the first granule after* t_1*.*

If $inclusion = subset$ *and* $P.sopC(c_1) < t_1$*, move* (c_1, c_2) *one granule further. If* $P.eopC(c_2) > t2$ *then 'which' is undefined.*

If $inclusion = overlaps$ *and* $t_2 < P.sopC(c_1)$ *then 'which' is undefined.*

If $inclusion = bigger\_part\_inside$ *and a larger part of the granule is before* t_1 *than after* t_1 *then move* (c_1, c_2) *one granule further. If* $P.eopC(c_2) > t2$ *then 'which' is undefined.*

Now count the number of granules from $(c1, c2)$ *until* n_2 *and return this number.* ∎

10 Summary

The basic ideas of the PartLib library for representing periodic temporal notions are explained in this paper. We use partitionings of the real numbers as basic mathematical structures. The partitionings can be labelled with symbolic names. The labels can be used for various purposes, in particular for defining *granules*, i.e. clusters of partitions which belong together semantically.

The approach proposed in this paper is a mix of algorithmic and symbolic specifications. The basic time units are realized as algorithmic partitionings where the details of the calendar system is hard-coded in the correction functions. All other periodic temporal notions are specified symbolically as regular, duration or folded partitionings. This seems to be a good compromise between the efficiency of compiled code for the difficult parts of calendar systems, and the flexibility of symbolic specifications for application specific parts.

The PartLib library is an open source C++ system. The details of its interface are explained in the appendix.

A The PartLib Interface

PartLib is an object oriented C++ implementation of the ideas presented in this paper. The interface to this library is presented here in a slightly abstracted way. We omit the technicalities of objects, pointers, references, const declarations etc. and just show the logical aspects of the interface. The technical details can be taken from the corresponding header files.

Nevertheless, certain technicalities are still important. One of them are the datatypes. We need 'int' (integers) and 'string' (strings) as basic datatypes. Floating point numbers are indicated by the type 'FLT'. This must be either 'float', 'double' or 'long double'. 'float' has only a precision of 6 digits, which is in general too small compared to a date like 2004/12/4/12/43/22, which has 13 digits. Therefore 'double' is recommended. Furthermore we use 'Rt' (for Reference time) and 'Co' (for Coordinate). They must both be integer datatypes. At least 64 bit integers are recommended. Bignumbers might be even better. Co and Rt can be the same or different. Since Co and Rt can be the same datatypes, PartLib cannot use overloaded functions in most cases. Therefore a number of function names have suffixes 'Co' and 'Rt'.

All the classes which are defined in PartLib can of course also be used as datatypes. These are in particular 'Label', 'Labelling', 'Duration', 'DateFormat', 'DateData', 'Partitioning', and the subclasses of 'Partitioning'.

The notation for the methods (functions) used in this appendix is

ResultType methodName(datatype$_1$ parameter$_1$,...)

'ResultType' is the datatype of the resulting value. If 'ResultType' is omitted then no result is returned by the method. 'datatype$_1$ parameter$_1$' etc. declares the datatypes of the parameters. 'datatype$_n$ parameter$_n$ = value' is also possible. It means that the default value for this parameter is 'value'.

NULL is the 'null pointer' which indicates that there is no object.

A.1 Repetition Patterns

Various classes defined below have constructor functions which accept part of their key parameters as strings. For example, the Labellings class has a constructor function which accepts the sequence of labels as a string of label names. These constructor functions make it easier to process user specified concepts by just passing the strings from the user input or from a specification file to the PartLib functions. Moreover, it allows the user to use shortcuts for repeating patterns of the same information.

As an example, where the usefulness of this becomes obvious, suppose we want to specify a bus timetable as a regular or duration partitioning. The concrete timetable is completely specified by listing the dates for Monday, declaring that this list is repeated 5 times (for Monday until Friday), then listing the dates for Saturday, and declaring that this second list is repeated twice. A compact notation for this would be something like '5*(list for Monday), 2*(list for Saturday)'. This compact notation can then be expanded by repeating the list for Monday five times, and concatenating it with two copies of the list for Saturday.

To enable this shortcut notation, we define a the formal language of *repetition patterns*.

Definition A.1 (Repetition Patterns) *Let B be a list of* base strings. *The base strings could for example be the string representation of integers, or the label names, or duration specifications. A* repetition pattern *R over B is the following formal language:*

1. *each element of B is a repetition pattern;*
2. *an expression $n * b$, where n is a non-negative integer and b is an element of B, is a repetition pattern;*
3. *a comma separated sequence of repetition patterns is repetition pattern;*
4. *an expression $n * (r)$, where n is a non-negative integer and r is a repetition pattern, is a repetition pattern.*

■

A repetition pattern can be expanded to a sequence of strings by repeating for $n * b$ the element b n times, and by repeating for expressions $n * (r)$ the expanded element r n times.

Example A.2 (for Repetition Pattern) *If B is a set of label names 'l1, l2, l3, l4, l5, gap' then '2*(l1, 2*l2, gap)' is a repetition pattern which expands to 'l1, l2, l2, gap, l1, l2, l2, gap'. '2*(l1, 2*(l2, gap))' expands to 'l1, l2, gap, l2, gap, l1, l2, gap, l2, gap'.* ■

A.2 The Label Class

A label is essentially a name (string). The Label class normalizese the representation of these strings such that two labels with the same name are stored only once.

Constructor Function

`Label(string name)`
: Constructs a new label with the given name. An error is thrown if the name is the empty string.

Public Instance Methods

`string name()`
: returns the name of the label in the current realm. name() throws an error if for the current realm no name is defined for the label.

`Label getLabel(string name)`
: If a label with this name exists already then this label is returned, otherwise a new label with this name is constructed and returned. getLabel throws an error if the name is the empty string.

`bool isGap()`
: returns true if the label is the gap label.

`bool notGap()`
: returns true if the label is not the gap label.

A.3 The Labelling Class

A labelling is just a vector of labels. The labelling class has no internal states. It therefore just manages individual labellings.

Constructor Functions

`Labelling(vector<Label> Labels)`
: constructs a new labelling from the given labels. (Def. 2.7)

`Labelling(string Labels)`
: 'Labels' is a repetition pattern over the set of label names (Def. A.1), which expands to a list of label names. If labels with these names do not yet exist, they are constructed. Different occurrences of the same name are mapped to the same label. The name 'gap' is mapped to the unique gap label. A new labelling is constructed with these labels. (Def. 2.7)

Public Instance Variables

`vector<Label> Labels`
: contains the labels

`int size`
: number of labels in the labelling

`int nGaps`
: number of gaps in the labelling

`bool longGranules`
: true if the labelling has granules consisting of more than one partition.

`bool longGranulesOrGaps`
: true if the labelling has long granules or gaps.

Public Instance Methods

`bool hasGaps()`
: returns true if there are gaps in the labelling

`bool validLabel(string name)`
: returns true if a label with this name exists in the labelling.

`Label label(Co coordinate)`
: returns the label for the given coordinate

`bool isGap(Co coordinate)`
: returns true if the label for the given coordinate is the gap label.

`bool notGap(Co coordinate)`
: returns true if the label for the given coordinate is not the gap label.

`Co nextNonGap(Co coordinate)`
: If the label at the given coordinate is not the gap label then 'coordinate' is returned. If it is the gap label then the coordinate of the next non-gap label is returned.

`Co previousNonGap(Co coordinate)`
: If the label at the given coordinate is not the gap label then 'coordinate' is returned. If it is the gap label then the coordinate of the previous non-gap label is returned.

`bool withinGranule(Co coordinate)`
: returns true if the label at the given coordinate is within a granule.

`Co nextGranule(Co coordinate)`
: returns the start coordinate of the next granule. The label at 'coordinate' may be within a granule or between granules.

`Co previousGranule(Co coordinate)`
: returns the end coordinate of the previous granule. The label at 'coordinate' may be within a granule or between granules.

`Co startOfGranule(Co coordinate)`
: If the label at 'coordinate' is within a granule then the start coordinate of this granule is returned. If the label is between two granules then the start coordinate of the next granule is returned.

`Co endOfGranule(Co coordinate)`
: If the label at 'coordinate' is within a granule then the end coordinate of this granule is returned. If the label is between two granules then the end coordinate of the previous granule is returned.

`Co lastCoordinate(Co coordinate)`
: returns the coordinate of the last non-gap label of the copy of the labelling at the given coordinate.

`Co lastCoordinate(Co coordinate)`
: returns the coordinate of the first non-gap label of the copy of the labelling at the given coordinate.

`int numberOfGaps(Co c1, Co c2)`
: returns the number of gap labels between coordinate c1 and c2 (both inclusive).

`granule(Co coordinate, Co c1, Co c2)`
: binds c1 to startOfGranule(coordinate) and c2 to endOfGranule(coordinate).

`Co nextGranuleWithLabel(Co coordinate, int n, string name)`
: If $n = 0$ then the given coordinate is just returned. If $n > 0$ then the start coordinate of the granule with the nth next occurrence of a label with the given name is returned. If $n < 0$ then the start coordinate of the granule with the nth previous occurrence of a label with the given name is returned.

`display()`
: prints the labelling to stdout (for debugging purposes.)

A.4 The Duration Class

A duration is a list of pairs (number, partitioning), together with a flag 'asGranule'. If this flag is true and there are labellings attached to the partitionings, then certain functions, in particular the shift function, takes the granules as basic units, and not the partitions.

Constructor Functions

`Duration(vector<pair<FLT,Partitioning> durations, bool asGranule=false)`
constructs a new duration.

`Duration(vector<FLT> shifts, vector<Partitioning> partitionings, bool asGranule=false)`
constructs a new duration by pairing the shifts and partitionings together.

`Duration(string duration, bool asGranule=false)`
constructs a new duration from a string representation '⟨number⟩ ⟨partitioning_name⟩, ...', for example, '3.5 month, -1 week, 2 hour'

Public Instance Methods

`Duration invert()`
constructs a new duration by inverting the given one

`append(Duration D)`
appends the duration D to the given one.

Example:	'3 week, 2 day'.append('1 hour, 3 minute')	→	'3 week, 2 day, 1 hour, 3 minute'
or:	'3 week, 2 day'.append('3 day, 3 minute')	→	'3 week, 5 day, 3 minute'
but:	'3 week, 2 day'.append('1 week')	→	'3 week, 2 day, 1 week'
and:	'1.5 week'.append('1.5 week')	→	'1.5 week, 1.5 week'

`Rt shiftOnce(Rt time, Partitioning P=NULL)`
shifts the time point 'time' by the given duration. If 'asGranule' is true then the time point is shifted by interpreting the partitionings as granules (see 'shiftAsGranule' below), otherwise they are interpreted as partitions. The underlying shift function is either 'shiftPartitiongsbyDate' or shiftGranules, if 'asGranules' = true.

`Rt shift(Rt time, FLT m=1.0, Partitioning P=NULL)`
shifts the time point 'time' by m times the given duration. m can be any positive of negative FLT number.

`vector<pair<Rt,Rt> > sections(Rt from, Rt to)`
splits the section [from,to[of the time axis into subsections $[(t_0, d_0), (t_1, d_1), ...]$, such that the duration causes a shift of d_i basic time units between the time points t_i and t_{i+1}. If the time shift is always the same value d_0 then only (from,d_0) is returned.

Relations

`bool operator<(Duration D1, Duration D2)`
This is a <-operator which compares two durations. It yields true if a shift by D1 is always shorter than a shift by D2. For example, '1 month, 1 day' < '4 week'. So far, PartLib contains only an approximative implementation of this relation: A finite number of time points is generated, and the shifts from these time points are compared. See also Section 2.4.

A.5 Class DateFormat

A date format is a list of partitions and a name. The name can be used to retrieve a previously created date format from the DateFormat class.

Constructor Functions

`DateFormat(string name, vector<Partitioning> partitionings)`
constructs a date format with the given name and partitionings.

`DateFormat(string name, string pnames)`
constructs a date format with the given name from the comma separated names of the partitionings.

Class Methods

`DateFormat getDateFormat(string name)`
returns a previously generated date format with this name. If the name is unknown then NULL is returned.

Public Instance Methods

`DateData date(Rt time, clear = true)`
turns a time point into dates (see below) of the given date format. If clear = true then trailing zeros are deleted, otherwise the date is exactly as long as the date format. (Def. 3.2)

A.6 Class DateData

A DateData object is essentially a date format and a sequence of integers, the dates.

Constructor Functions

`DateData(DateFormat dF, vector<Co> data)`
constructs a DateData object for the given date format from the data.

`DateData(DateFormat dF, string dD, string separator = "/")`
constructs a DateData object for the given date format from a string representation of the data. The data must be a sequence of positive or negative integers, separated by the given separator. An error is thrown if the string cannot be parsed properly.

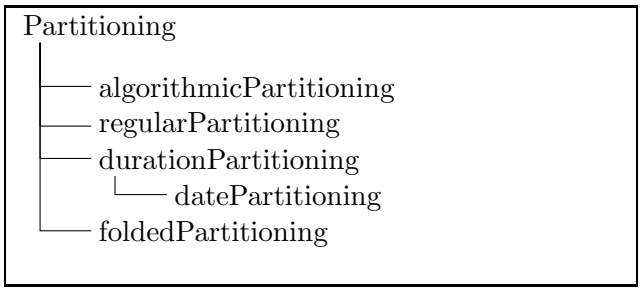
Public Instance Methods

`Rt TimePoint()`
turns the DateData object into the corresponding time point. (Def. 3.3)

`append(vector<Co> date, Co extra)`
appends 'date' at the end of the given date and adds 'extra' to date[0].

A.7 The Partitioning Class

The 'Partitioning' class consists of the abstract class 'Partitioning' and a few subclasses. The class diagram is:



The partitionings are parameterized with an instance of a class 'CalContext'. This class is not defined in PartLib. It must provide the following methods:

- *int resolution*(): which returns a positive integer. 1 means, the basic time unit are seconds. 1000 means, the basic time unit are milliseconds.
- *int timezone*() returns the timezone as an integer.
- *leapSeconds*() returns a LeapSeconds object, which has the two methods $globalCorrection(grt)$ (lsG in Def. 7.1) and $localCorrection(lrt)$ (lsL in Def. 7.1).

A.7.1 The Abstract Class 'Partitioning'

This class is essentially the interface to the PartLib library. Except for the constructor functions of the subclasses, one should use only the public methods from this class.

General Methods

`name`
: is the public instance variable which holds the name of the partitioning.

`static Partitioning getPartitioning(string name)`
: returns the partitioning with the given name, or NULL if this name is unknown.

`static vector<Partitioning> allPartitionings();`
: returns a vector of all partitionings.

Coordinate Computations

`Rt startOfPartitionRt(Rt time)`
: maps the given time point to the start time of the partition containing this point. (Def. 2.3)

`Rt endOfPartitionRt(Rt time)`
: maps the given time point to the end time of the partition containing this point. (Def. 2.3)

`Rt startOfPartitionCo(Co coordinate)`
: maps the given coordinate to the start time of the partition with this coordinate (cf. *sopC*). (Def. 2.3)

`Rt endOfPartitionCo(Co coordinate)`
: maps the given coordinate to the end time of the partition with this coordinate (cf. *eopC*). (Def. 2.3)

`Co partitionCoordinate(Rt time)`
: maps a time point to the coordinate of the partition containing this point (cf. *pc*). (Def. 2.3)

Labels and Granules

`setLabelling(Labelling L)`
: attaches the labelling L to the given partitioning. This method throws an error for folded partitionings.

`bool validLabel(string label)`
: returns true if the label occurs in the labelling.

`Labelling labellingCo(Co coordinate)`
: returns the labelling which is valid for the given coordinate and NULL if there is none. This is non-trivial only for folded partitionings.

`Labelling labellingRt(Rt time)`
: returns the labelling which is valid for the given time, and NULL if there is none. This is non-trivial only for folded partitionings.

`labelCo(Co coordinate)`
: returns the label which is attached to the given coordinate, and NULL if there is none.

`labelRt(Rt time)`
: returns the label which is attached to the partition containing this time, and NULL if there is none.

`bool isGap(Co coordinate)`
: returns true if the label at this gap is the gap label.

`bool notGap(Co coordinate)`
: returns true if the label at this gap is not the gap label.

`bool hasGaps(Co coordinate)`
: returns true if the labelling which is valid for this coordinate has gaps. The coordinate plays only a role for folded partitionings.

`bool withinGranuleCo(Co coordinate)`
: true if the coordinate is within a granule.

`bool withinGranulRt(Rt time)`
: true if the time point is within a granule.

`bool closestGranuleCo(Co coordinate, Co c1, Co c2)`
: binds c1 and c2 to the start and end coordinates of the granule which is closest to 'coordinate'. The result value is true if the coordinate is within a granule. (Def. 2.11)

`bool closestGranuleRt(Rt time, Co c1, Co c2)`
: binds c1 and c2 to the start and end coordinates of the granule which is closest to the given time point. The result value is true if the time point is within a granule. (Def. 2.11)

Local Length of Partitions and Granules

`Rt lengthOfPartitionCo(Co coordinate)`
: returns the length of the partition with the given coordinate.

`Rt lengthOfPartitionRt(Rt time)`
: returns the length of the partition containing the given time point.

`FLT lengthInPartitions(Rt t1, Rt t2)`
: returns the length of the interval [t1,t2[in terms of the length of the partitions of the given partitioning. (Def. 2.5)

`FLT lengthInGranules(Rt t1, Rt t2)`
: returns the length of the interval [t1,t2[in terms of the length of the granules of the given partitioning. (Def. 2.12)

Global Length of Partitions and Granules

`setShortestPartition(Rt length)`
: overrides the randomized computation of partition length by just declaring the shortest partition length.

`setLongestPartition(Rt length)`
: overrides the randomized computation of partition length by just declaring the longest partition length.

`Rt shortestPartition()`
: returns the length of the shortest partition.

`Rt longestPartition()`
: returns the length of the longest partition.

`Rt shortestGranule()`
: returns the length of the shortest granule.

`Rt longestGranule()`
: returns the length of the longest granule.

`bool hasShorterPartitionsThan(Partitioning Q)`
: returns true if the given partitioning has shorter partitions than partitioning Q. (Def. 2.13)

`bool hasShorterGranulesThan(Partitioning Q)`
: returns true if the given partitioning has shorter granules than partitioning Q. (Def. 2.13)

Shift of Time Points

`Rt shiftPartitionsbyLength(Rt time, FLT m, DateFormat dF = NULL, Partitioning P = NULL`
: shifts the time point by the length of m partitions. If a date format dF and a partitioning P is given then the shifted time point is moved again such that the dates of the shifted time and the original time are kept the same from the partitioning P onwards. For example, if the date format is year/month/day/hour/minute/second, and P = hour, then the shifted time has the same hour/minute/second date as the original time. (Def. 4.2)

`shiftPartitionsbyLength(Rt time, FLT m, string dF, string P)`
: This method is almost identical to the above method. The only difference is that the date format and the partitioning are given by their names, and not by the pointers. (Def. 4.2)

`Rt shiftPartitionsbyDate(Rt time, FLT m, Partitioning P = NULL`
: shifts the time point by m partitions using the dates for the shift. If a partitioning P is given then the shifted time point is moved again such that the dates of the shifted time and the original time are kept the same from the partitioning P onwards. (Def. 8.5)

`Rt shiftPartitionsbyDate(Rt time, FLT m, string P`
: This method is almost identical to the above method. The only difference is that the partitioning is given by its names, and not by the pointer. (Def. 8.5)

`Rt granules2Partitions(Rt time, FLT m)`
: turns the number `m` of granules into a number `n` of partitions. `m` can be a positive or negative fractional number. (Sect. 4.3)

`Co nextGranuleWithLabel(Co coordinate, int n, string label)`
: If $n = 0$ then 'coordinate' is returned. If $n > 0$ then the coordinate of the start of the nth next granule with the given label name is returned. If $n < 0$ then the coordinate of the end of the nth previous granule with the given label name is returned.

Service Functions

`Co which(Rt t, Partitioning Q, Inclusion incl, bool asGranule, bool valid)`
: realizes the 'which' function. 'valid' is also an output parameter. $valid = false$ means that the conditions could not be met, and therefore the result is only approximative. Usually, this means the result is either 0 or the result of $which(t', \ldots)$ where t' is the smallest time point after t, for which a valid result can be obtained. (Def. 9.1)

`Co which(Rt t, string Q, Inclusion incl, bool asGranule, bool valid)`
: Is like the above function. The only difference is that the name of the partitioning Q is given, instead of its address. (Def. 9.1)

Structural Relations

`bool includesPartitions(Partitioning Q)`
: returns true if each partition of the given partition is a subset of a partition of Q. (Def. 2.13)

`bool includesGranules(Partitioning Q)`
: returns true if each granule of the given partition is a subset of a granule of Q. (Def. 2.13)

`bool subsetGranule(Rt time, Co coP, Partitioning Q, Co coQ)`
: returns true if the P-granule at coordinate coP is a subset of the Q-granule at coordinate coQ. (Def. 2.13)

`bool operator<(Partitioning P, Partitioning Q)`
: $P < Q$ is just short for P.hasShorterPartitionsThan(Q). (Def. 2.13)

Boundaries

`bool leftBounded`
: is a public instance variable, which is true when a left boundary is defined.

`Rt leftBoundary`
: is an instance variable which holds the left boundary.

`bool rightBounded`
: is a public instance variable, which is true when a right boundary is defined.

`Rt rightBoundary`
: is an instance variable which holds the right boundary.

`void setLeftBoundary(Rt time)`
: sets the left boundary.

`void setRightBoundary(Rt time)`
: sets the right boundary.

A.7.2 Subclasses of the Partitioning class

Only the constructor functions for the subclasses are in the interface. All other methods are to be called via the Partitioning class interface.

`algorithmicPartitioning(string name, int avl, Co2Rt cf, Rt offset, CalContext C, int repetitions, Labelling L=NULL)`
: constructs an arithmetic partitioning. 'name' is the name of the partitioning, 'avl' is the average length of the partitions. 'cf' is the correction function. 'offset' is the offset of the partition with coordinate 0. 'C' is the context object, 'repetitions' is the parameter for the randomized checks, and 'L' is a labelling. (Def. 8.1)

`regularPartitioning(string name, Rt anchor, vector<FLT> sizes, Partitioning U, CalContext C, bool asGranule, Labelling L=NULL)`
: constructs a regular partitioning.'name' is the name of the partitioning. 'anchor' is the anchor time. 'sizes' specifies the lengths of the partitions in terms of the partitioning 'U'. 'C' is the context object. 'asGranule = true' causes the sizes to be interpreted as granule length. 'L' is a labelling. (Def. 8.10)

`regularPartitioning(string name, Rt anchor, string sizes, string U, CalContext C, bool asGranule, Labelling L=NULL)`
: is almost the same as the above constructor function. The difference is that the sizes are given as repetition pattern over numbers (Def. A.1) and the partitioning 'U' is given by its name. This function throws an error if 'U' is unknown or if 'sizes' cannot be parsed. (Def. 8.10)

`durationPartitioning(string name, Rt anchor, vector<Duration> durations, CalContext C, Labelling L=NULL)`
: constructs a duration partitioning. 'name' is the name of the partitioning. 'anchor' is the anchor time. 'durations' specifies the lengths of the partitions as durations 'C' is the context object. 'L' is a labelling. (Def. 8.7)

`durationPartitioning(string name, Rt anchor, string durations, CalContext C, Labelling L=NULL)`

constructs a duration partitioning. 'name' is the name of the partitioning. 'anchor' is the anchor time. 'durations' is a repetition pattern over duration specifications in quotes (see the constructor function 'Duration'). An example is '2*("1 month, 1 week", "2 day, -1 hour")'. The pattern expands to a sequence of durations. They specify the lengths of the partitions as durations 'C' is the context object. 'L' is a labelling. (Def. 8.7)

`datePartitioning(string name, DateFormat dateFormat, vector<vector<Co>> dates, CalContext C, Labelling L=NULL)`

constructs a date partitioning. 'name' is the name of the partitioning. 'dateFormat' is the date format for the dates. 'dates' is the vector of dates. 'C' is the context object. 'L' is a labelling. (Def. 8.16)

`datePartitioning(string name, DateFormat dateFormat, string dates, CalContext C, Labelling L=NULL)`

This constructor function is almost like the above function. The difference is that the dates are given as ' '-separated dates (which are '/'-separated integers). (Def. 8.16)

`foldedPartitioning(string name, Partitioning framePartitioning, vector<Partitioning> componentPartitionings, bool constant, CalContext C)`

constructs a folded partitioning. 'name' is the name of the partitioning. 'framePartitioning' is the frame partitioning and 'componentPartitionings' is the vector of component partitionings. The 'constant' flag indicates whether this is a constant partitioning. This is not checked! If a folded partitioning is declared constant, but is not, the results of all computations are unpredictable. (Def. 8.17)

`foldedPartitioning(string name, Partitioning framePartitioning, string componentPartitionings, bool constant, CalContext C)`

This constructor function is almost like the above function. The difference is that the component partitionings are given as repetition pattern over the partitioning names (Def. A.1). This function throws an error if partitioning names are unknown. (Def. 8.17)

References

[1] C. Bettini and R.D.Sibi. Symbolic representation of user-defined time granularities. *Annals of Mathematics and Artificial Intelligence*, 30:53–92, 2000. Kluwer Academic Publishers.

[2] Claudio Bettini, Curtis E. Dyreson, William S. Evans, Richard T. Snodgrass, and X. Sean Wang. *Temporal Databases, Rreseach and Practice*, volume 1399 of *LNCS*, chapter A Glossary of Time Granularity Concepts, pages 406–413. Springer Verlag, 1998.

[3] Claudio Bettini, Sushil Jajodia, and Sean X. Wang. *Time Granularities in Databases, Data Mining and Temporal Reasoning*. Springer Verlag, 2000.

[4] Claudio Bettini, Sergio Mascetti, and X. Sean Wang. Mapping calendar expressions into periodical granularities. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 87–95, Los Alamitos, California, 2004. IEEE.

[5] François Bry, Frank-André Rieß, and Stephanie Spranger. CaTTS: Calendar Types and Constraints for Web Applications. research report, PMS-FB-2004-24 PMS-FB-2004-24, Institute for Informatics, University of Munich, 2004.

[6] Diana R. Cukierman. *A Formalization of Structured Temporal Objects and Repetition*. PhD thesis, Simon Franser University, Vancouver, Canada, 2003.

[7] Diana R. Cukierman and James P. Delgrande. Expressing time intervals and repetition within a formalization of calendars. *Computational Intelligence*, 14(4):563–597, 1998.

[8] Diana R. Cukierman and James P. Delgrande. The SOL time theory: A formalization of structured temporal objects and repetition. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 71–34, Los Alamitos, California, 2004. IEEE.

[9] Nachum Dershowitz and Edward M. Reingold. *Calendrical Calculations*. Cambridge University Press, 1997.

[10] Curtis E. Dyreson, Wikkima S. Evans, Hing Lin, and Richard T. Snodgrass. Efficiently supporting temporal granularities. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):568–587, 2000.

[11] Lavinia Egidi and Paolo Terenziani. A lattice of classes of user-defined symbolic periodicities. In C. Combi and G. Ligozat, editors, *Proc. of the 11th International Symposium on Temporal Representation and Reasoning*, pages 13–20, Los Alamitos, California, 2004. IEEE.

[12] I.A. Goralwalla, Y. Leontiev, M.T. Ozsu, D. Szafron, and C. Combi. Temporal granularity: Completing the picture. *Journal of Intelligent Information Systems*, 16(1):41–63, 2001.

[13] Nick Kline, Jie Li, and Richard Snodgrass. Specifying multiple calendars, calendric systems and field tables and functions in timeadt. Technical Report TR-41, Time Center Report, May 1999.

[14] B. Leban, D. Mcdonald, and D.Foster. A representation for collections of temporal intervals. In *Proc. of the American National Conference on Artificial Intelligence (AAAI)*, pages 367–371. Morgan Kaufmann, Los Altos, CA, 1986.

[15] B. Leban, D. D. McDonald, and D. R. Forster. A representation of collections of temporal intervals. In *Proc. of AAAI'86*, pages 367–371, 1986.

[16] M. Niezette and J. Stevenne. An efficient symbolic representation of periodic time. In *Proc. of the first International Conference on Information and Knowledge Management*, volume 752 of *Lecture Notes in Computer Science*, pages 161–169. Springer Verlag, 1993.

[17] Peng Ning, X. Sean Wang, and Sushil Jajodia. An algebraic representation of calendars. *Annals of Mathematics and Artificial Intelligenc*, 36(1-2):5–38, September 2002. Kluwer Academic Publishers.

[18] Hans Jürgen Ohlbach. Computational treatment of temporal notions – the CTTN-system. Research Report PMS-FB-2005-30, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-30.

[19] Hans Jürgen Ohlbach. Fuzzy time intervals – the FuTI-library. Research Report PMS-FB-2005-26, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-26.

[20] Hans Jürgen Ohlbach. GeTS – a specification language for geo-temporal notions. Research Report PMS-FB-2005-29, Inst. für Informatik, LFE PMS, University of Munich, June 2005. URL: http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-29.

[21] Michael D. Soo and Richard T. Snodgrass. Mixed calendar query language support for temporal constants. Technical Report TR 92-07, Dept. of Computer Science, Univ. of Arizona, February 1992.